Colas de prioridad (priority queues)

mat-151

Colas de prioridad

Es una cola a cuyos elementos se les asigna una prioridad.

Estos elementos son procesados de la siguiente manera:

- el elemento de mayor prioridad es procesado primero.
- dos elementos con la misma prioridad son procesados según el orden en que fueron introducidos en la cola.

Colas de prioridad: ADT colas vs. colas de prioridad

COLAS

- **insert:** agregar un elemento al final de la cola.
- **remove:** eliminar el elemento al frente de la cola.
- **getFront:** regresar el elemento al frente de la cola (sin eliminarle).

COLAS DE PRIORIDAD

insert: agregar un elemento, con una prioridad asociada, en cualquier posición en la cola.

eliminar: eliminar el primer elemento en la cola con la mayor prioridad.

getHighestPriority: regresar el primer elemento en la cola con la mayor prioridad.

Colas de prioridad: aplicaciones

- Simulación basada en eventos, procesar el siguiente evento que ocurra (orden cronológico)
- planificación de trabajos computacionales (prioridad de los usuarios, tareas mas cortas);
- computación numérica (error mas grande primero);
- encontrar el k-ésimo elemento mas grande;

- en un hospital, atender al paciente mas grave primero.
- filtros anti-spam, ordenamiento de e-mails.
- tareas de un robot ordenadas por prioridad...

Colas de prioridad - 1. arreglos no-ordenados

los elementos se insertan y eliminan del final del arreglo, como en una pila, **no** se mantienen en orden de prioridad.

el método insert agrega un elemento al final de la cola.

el metodo remove:

- encuentra al elemento con mayor prioridad.
- elimina el nodo reemplazandole con el nodo al final de la cola.

```
class Priority_Queue
{
    private:
        Item *pq;
        int N;
    public:
        Priority_Queue( int maxN ){
            pq = new Item[maxN];
            N=0;
        }
        int empty() const {
            return N == 0;
        }
        void insert(Item item){
            pq[N++] = item;
        }
        Item getmax(){
            int max=0;
            for ( int j=1; j<N; j++ )
                if( pq[max] < pq[j] )
                    max = j;
            exch( pq[max], pq[N-1] );
            return pq[--N];
        }

};
```

Heaps y Heapsort

mat-151

Montículos (Heaps)

Montículos (Heaps)

- Estructura de datos que soporta de manera eficiente las operaciones básicas de las colas de prioridad.

Montículos (Heaps)

- Estructura de datos que soporta de manera eficiente las operaciones básicas de las colas de prioridad.

- Conjunto de elementos con la propiedad que cada llave tiene la garantía de ser más grande (max heap) que las llaves en dos posiciones especificas:

Montículos (Heaps)

- Estructura de datos que soporta de manera eficiente las operaciones básicas de las colas de prioridad.
- Conjunto de elementos con la propiedad que cada llave tiene la garantía de ser más grande (max heap) que las llaves en dos posiciones especificas:
  - llave(A) $\geq$ llave(B).

Montículos (Heaps)

- Estructura de datos que soporta de manera eficiente las operaciones básicas de las colas de prioridad.
- Conjunto de elementos con la propiedad que cada llave tiene la garantía de ser más grande (max heap) que las llaves en dos posiciones especificas:
  - llave(A) $\geq$ llave(B).
- Cada una de estas dos llaves tienen la misma garantía.

Montículos (Heaps)

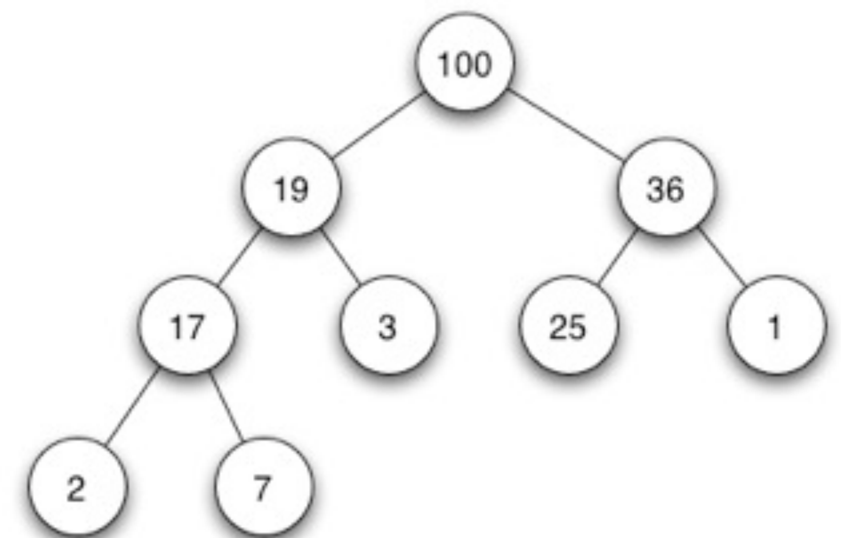
- Estructura de datos que soporta de manera eficiente las operaciones básicas de las colas de prioridad.
- Conjunto de elementos con la propiedad que cada llave tiene la garantía de ser más grande (max heap) que las llaves en dos posiciones especificas:

  •llave(A) $\geq$ llave(B).

- Cada una de estas dos llaves tienen la misma garantía.
- ¿A que estructura de datos nos hace pensar?

Montículos (Heaps)

- Estructura de datos que soporta de manera eficiente las operaciones básicas de las colas de prioridad.
- Conjunto de elementos con la propiedad que cada llave tiene la garantía de ser más grande (max heap) que las llaves en dos posiciones especificas:
    - llave(A) ≥ llave(B).
- Cada una de estas dos llaves tienen la misma garantía.
- ¿A que estructura de datos nos hace pensar?



Montículos binarios (Binary heaps)

Montículos binarios (Binary heaps)

- Representación en un arreglo de un árbol binario en orden de montículo.

Montículos binarios (Binary heaps)

- Representación en un arreglo de un árbol binario en orden de montículo.
- Orden de montículo:

Montículos binarios (Binary heaps)

- Representación en un arreglo de un árbol binario en orden de montículo.
- Orden de montículo:
  - Llaves en los nodos

Montículos binarios (Binary heaps)

- Representación en un arreglo de un árbol binario en orden de montículo.
- Orden de montículo:
  - Llaves en los nodos
  - No mas pequeño que las llaves de sus hijos

Montículos binarios (Binary heaps)

- Representación en un arreglo de un árbol binario en orden de montículo.
- Orden de montículo:
  - Llaves en los nodos
  - No mas pequeño que las llaves de sus hijos
- Representación de árbol con arreglo

Montículos binarios (Binary heaps)

- Representación en un arreglo de un árbol binario en orden de montículo.
- Orden de montículo:
  - Llaves en los nodos
  - No mas pequeño que las llaves de sus hijos
- Representación de árbol con arreglo
  - Toma los nodos en orden por nivel

Montículos binarios (Binary heaps)

- Representación en un arreglo de un árbol binario en orden de montículo.
- Orden de montículo:
  - Llaves en los nodos
  - No mas pequeño que las llaves de sus hijos
- Representación de árbol con arreglo
  - Toma los nodos en orden por nivel
  - No necesita links explícitos

Montículos binarios (Binary heaps)

- Representación en un arreglo de un árbol binario en orden de montículo.

i	0	1	2	3	4	5	6	7	8	9	10	11
a[i]	-	T	S	R	P	N	O	A	E	I	H	G

- Orden de montículo:
  - Llaves en los nodos
  - No mas pequeño que las llaves de sus hijos
- Representación de árbol con arreglo
  - Toma los nodos en orden por nivel
  - No necesita links explícitos

Montículos binarios (Binary heaps)

- Representación en un arreglo de un árbol binario en orden de montículo.
- Orden de montículo:
  - Llaves en los nodos
  - No mas pequeño que las llaves de sus hijos
- Representación de árbol con arreglo
  - Toma los nodos en orden por nivel
  - No necesita links explícitos

i	0	1	2	3	4	5	6	7	8	9	10	11
a[i]	-	T	S	R	P	N	O	A	E	I	H	G

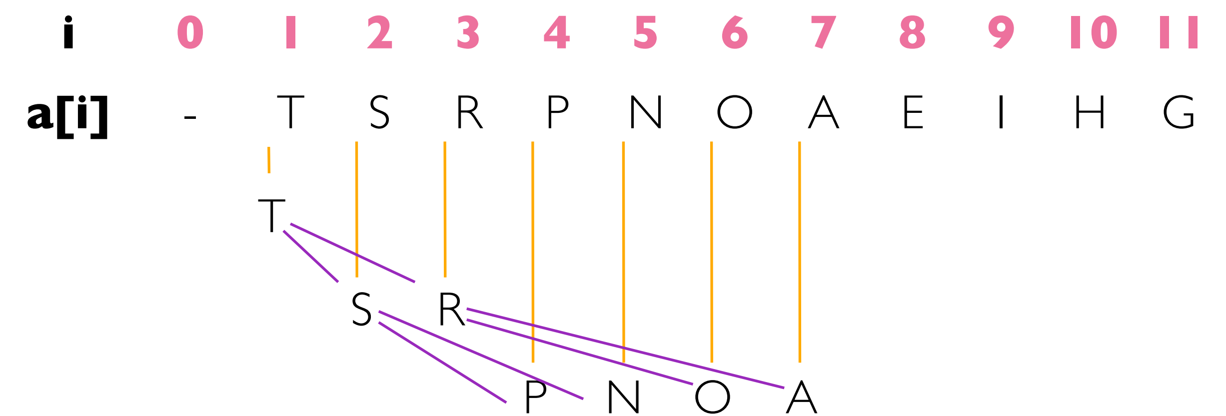
T

Montículos binarios (Binary heaps)

- Representación en un arreglo de un árbol binario en orden de montículo.
- Orden de montículo:
  - Llaves en los nodos
  - No mas pequeño que las llaves de sus hijos
- Representación de árbol con arreglo
  - Toma los nodos en orden por nivel
  - No necesita links explícitos

Montículos binarios (Binary heaps)

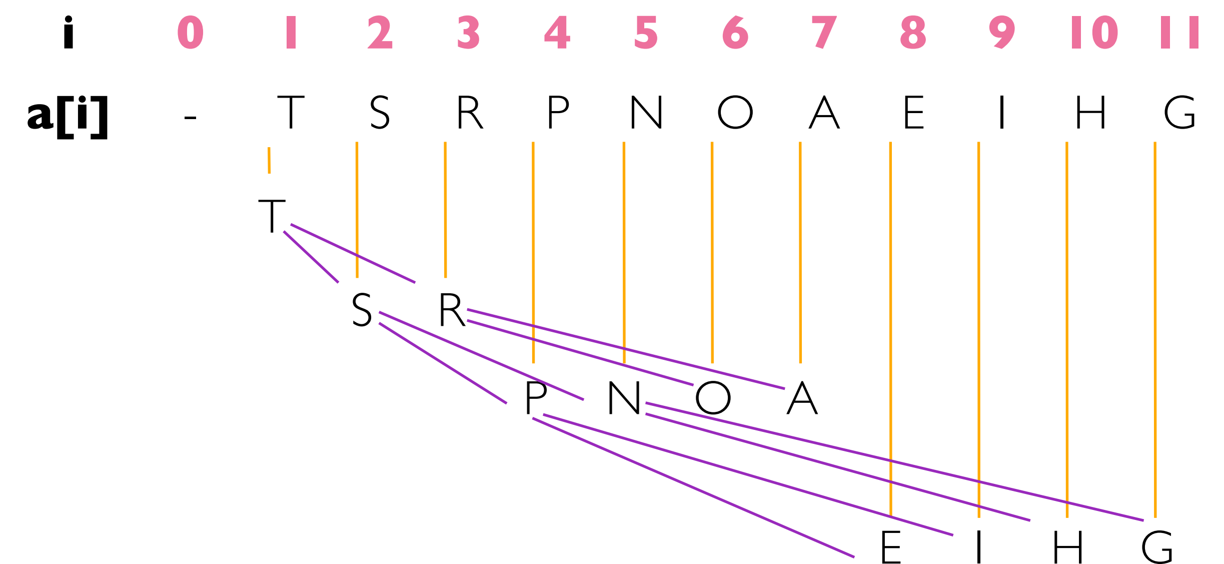
- Representación en un arreglo de un árbol binario en orden de montículo.
- Orden de montículo:
  - Llaves en los nodos
  - No mas pequeño que las llaves de sus hijos
- Representación de árbol con arreglo
  - Toma los nodos en orden por nivel
  - No necesita links explícitos



Friday, April 7, 17

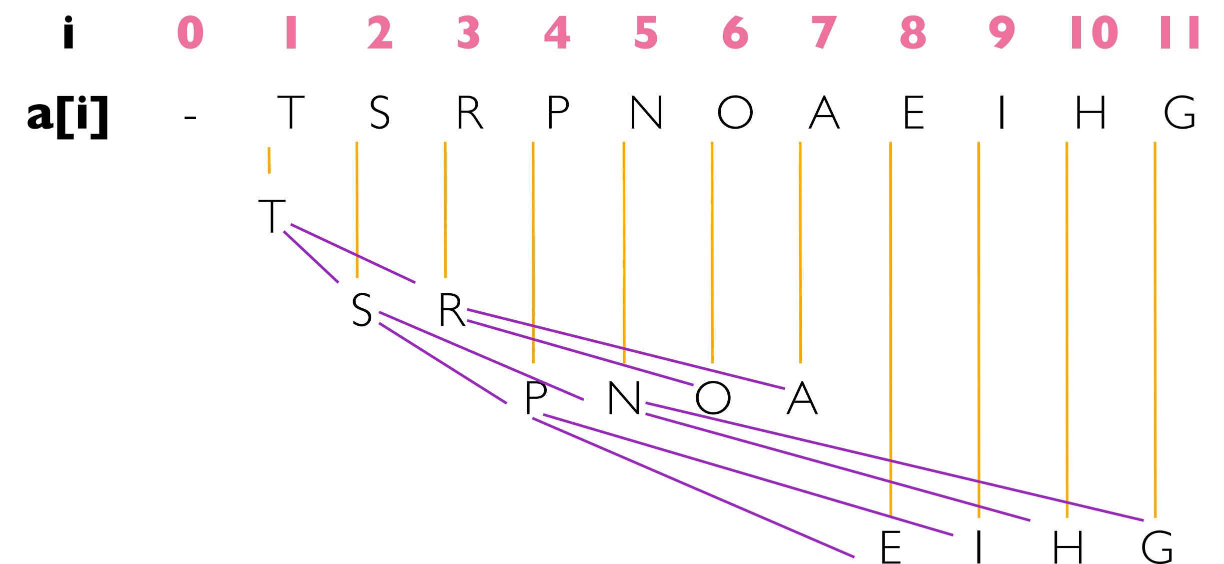
Montículos binarios (Binary heaps)

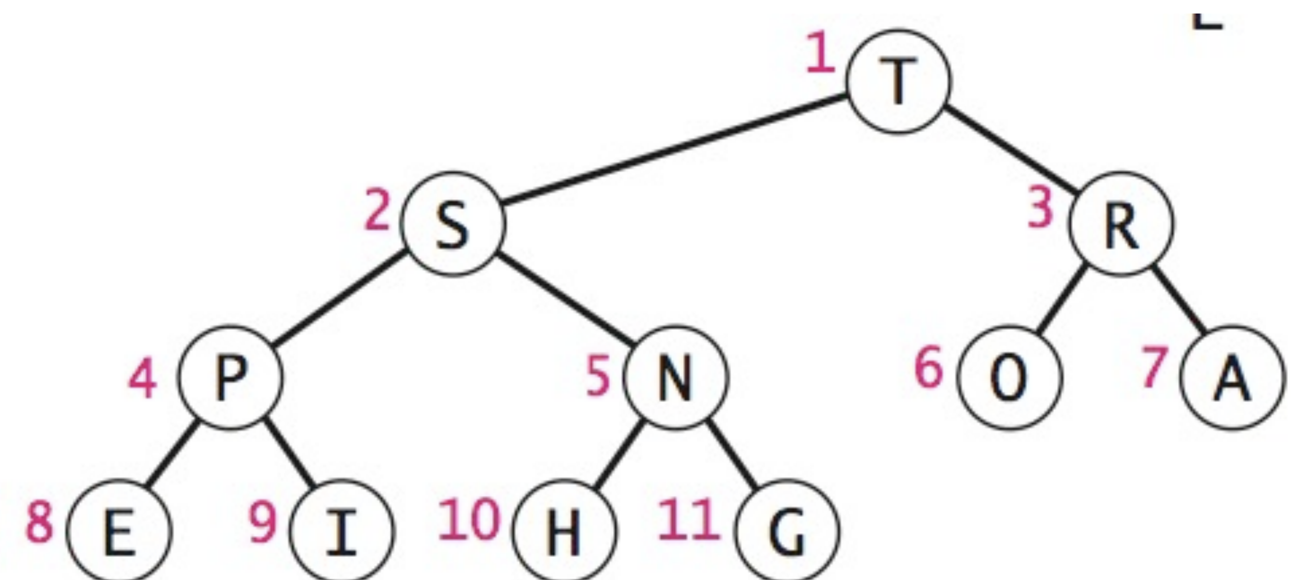
- Representación en un arreglo de un árbol binario en orden de montículo.
- Orden de montículo:
  - Llaves en los nodos
  - No mas pequeño que las llaves de sus hijos
- Representación de árbol con arreglo
  - Toma los nodos en orden por nivel
  - No necesita links explícitos



Montículos binarios (Binary heaps)

- Representación en un **arreglo** de un árbol binario en orden de montículo.
- **Orden** de **montículo**:
  - Llaves en los nodos
  - No mas pequeño que las llaves de sus hijos
- **Representación** de árbol con **arreglo**
  - Toma los nodos en orden por nivel
  - No necesita links explícitos





Montículos binarios (Binary heaps)

Montículos binarios (Binary heaps)

- **Propiedad 1:**
  - La llave mas grande es la raíz.
- **Propiedad 2:**
  - Puede usar los índices del arreglo para moverse a lo largo del árbol.
    - Los índices se cuentan a partir de 1.
    - El padre del nodo k esta en el índice $k/2$.
    - Los hijos del nodo en k están en $2k$ y $2k+1$.

Montículos binarios (Binary heaps)

Montículos binarios (Binary heaps)

- A veces conviene una implementación de arreglo y no de listas ligadas.

Montículos binarios (Binary heaps)

- A veces conviene una implementación de arreglo y no de listas ligadas.
- Con una lista ligada necesitaríamos 3 links asociados a cada llave (uno a su padre y uno a cada hijo).

Montículos binarios (Binary heaps)

- A veces conviene una implementación de arreglo y no de listas ligadas.
- Con una lista ligada necesitaríamos 3 links asociados a cada llave (uno a su padre y uno a cada hijo).
- Cuando son árboles completos con estructura conviene una implementación de arreglo.

Montículos binarios (Binary heaps)

- A veces conviene una implementación de arreglo y no de listas ligadas.
- Con una lista ligada necesitaríamos 3 links asociados a cada llave (uno a su padre y uno a cada hijo).
- Cuando son árboles completos con estructura conviene una implementación de arreglo.
- Un montículo permite implementar todas las operaciones de una cola de prioridad (excepto ***join***) en un tiempo de ejecución logarítmico en el peor caso.

Montículos binarios (Binary heaps)

- A veces conviene una implementación de arreglo y no de listas ligadas.
- Con una lista ligada necesitaríamos 3 links asociados a cada llave (uno a su padre y uno a cada hijo).
- Cuando son árboles completos con estructura conviene una implementación de arreglo.
- Un montículo permite implementar todas las operaciones de una cola de prioridad (excepto ***join***) en un tiempo de ejecución logarítmico en el peor caso.
- Estas implementaciones involucran un recorrido en el árbol (moviendo de padre a hijo hacia abajo o de hijo a padre hacia arriba, pero no intercambiar direcciones).

Algoritmos con montículos

Algoritmos con montículos

- Los algoritmos de colas de prioridad con montículos funcionan primero haciendo una modificación que puede violar el orden de éste.

Algoritmos con montículos

- Los algoritmos de colas de prioridad con montículos funcionan primero haciendo una modificación que puede violar el orden de éste.
- Esta modificación obliga a hacer un recorrido por el árbol para verificar la condición de orden.

Algoritmos con montículos

- Los algoritmos de colas de prioridad con montículos funcionan primero haciendo una modificación que puede violar el orden de éste.
- Esta modificación obliga a hacer un recorrido por el árbol para verificar la condición de orden.
- A este proceso se le conoce como *heapifying* o arreglo de montículo.

Algoritmos con montículos

- Los algoritmos de colas de prioridad con montículos funcionan primero haciendo una modificación que puede violar el orden de éste.
- Esta modificación obliga a hacer un recorrido por el árbol para verificar la condición de orden.
- A este proceso se le conoce como *heapifying* o arreglo de montículo.
- Dos casos:

Algoritmos con montículos

- Los algoritmos de colas de prioridad con montículos funcionan primero haciendo una modificación que puede violar el orden de éste.
- Esta modificación obliga a hacer un recorrido por el árbol para verificar la condición de orden.
- A este proceso se le conoce como *heapifying* o arreglo de montículo.
- Dos casos:
  - cuando se aumenta la prioridad de un nodo (o se agrega un nodo al final del montículo): recorrido hacia arriba.

Algoritmos con montículos

- Los algoritmos de colas de prioridad con montículos funcionan primero haciendo una modificación que puede violar el orden de éste.
- Esta modificación obliga a hacer un recorrido por el árbol para verificar la condición de orden.
- A este proceso se le conoce como *heapifying* o arreglo de montículo.
- Dos casos:
    - cuando se aumenta la prioridad de un nodo (o se agrega un nodo al final del montículo): recorrido hacia arriba.
    - cuando se disminuye la prioridad de un nodo (o se reemplaza la raíz con otro nodo): recorrido hacia abajo.

Promoción de **un** nodo (recorrido hacia arriba)

Promoción de **un** nodo (recorrido hacia arriba)

- un nodo tiene un valor de llave más grande que su padre:

Promoción de **un** nodo (recorrido hacia arriba)

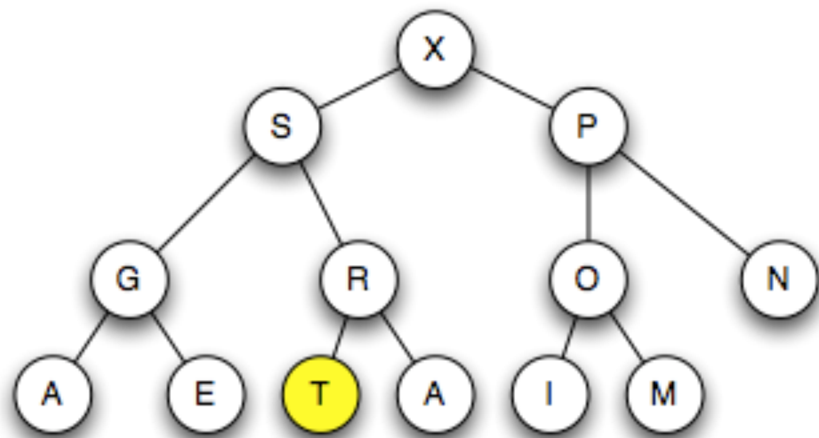
- un nodo tiene un valor de llave más grande que su padre:
  - intercambiar al nodo con su padre.

Promoción de **un** nodo (recorrido hacia arriba)

- un nodo tiene un valor de llave más grande que su padre:
  - intercambiar al nodo con su padre.
  - repetir el proceso hasta verificar orden.

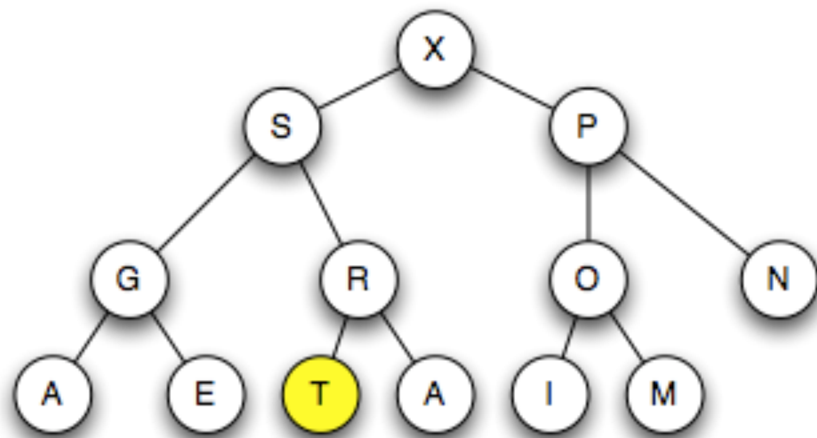
Promoción de **un** nodo (recorrido hacia arriba)

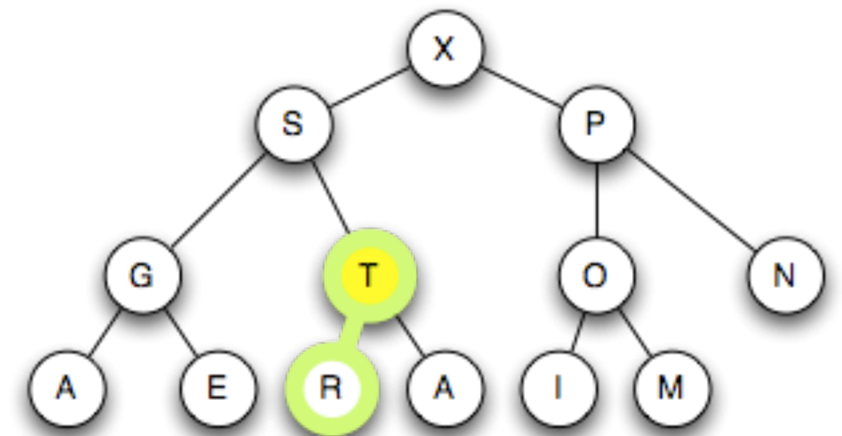
- un nodo tiene un valor de llave más grande que su padre:
  - intercambiar al nodo con su padre.
  - repetir el proceso hasta verificar orden.



Promoción de **un** nodo (recorrido hacia arriba)

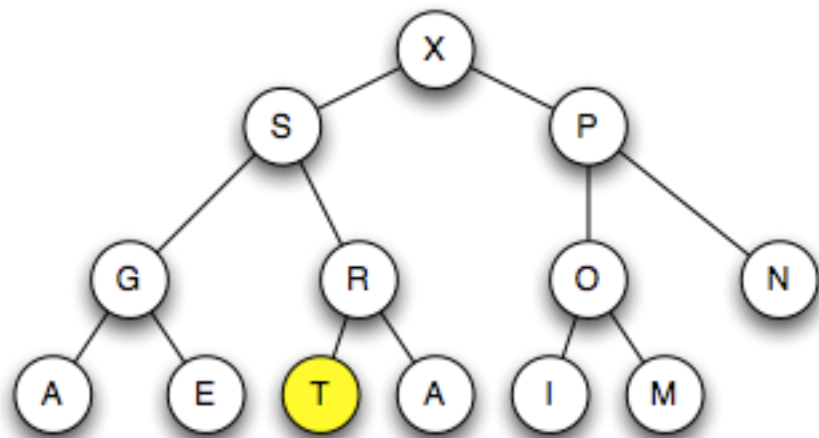
- un nodo tiene un valor de llave más grande que su padre:
  - intercambiar al nodo con su padre.
  - repetir el proceso hasta verificar orden.

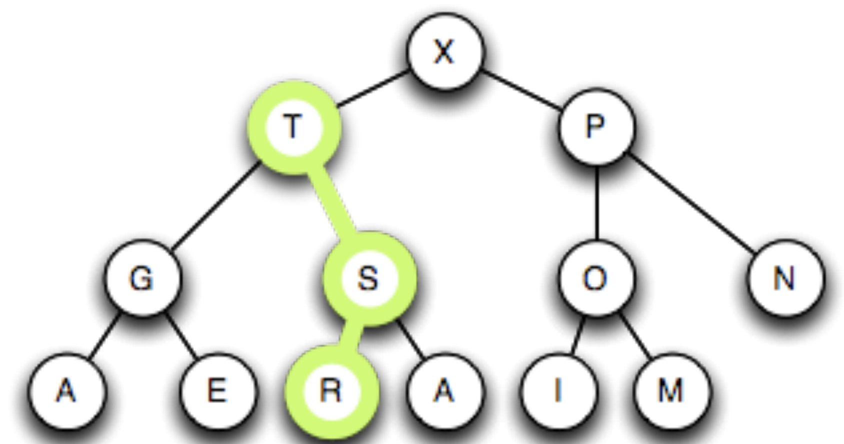




Promoción de **un** nodo (recorrido hacia arriba)

- un nodo tiene un valor de llave más grande que su padre:
  - intercambiar al nodo con su padre.
  - repetir el proceso hasta verificar orden.

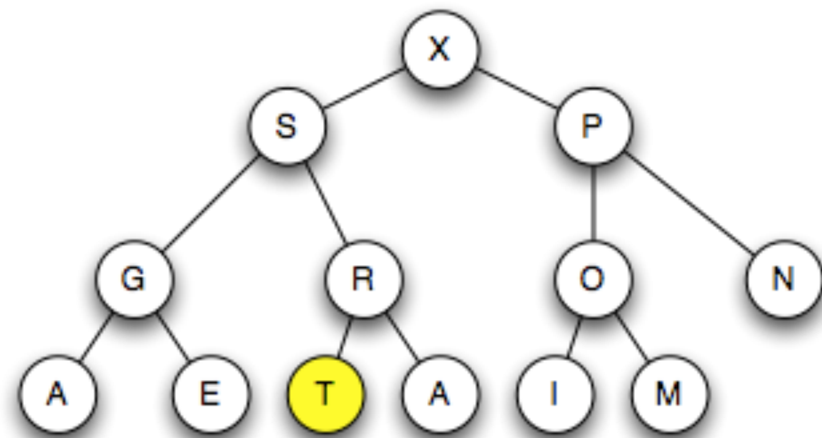


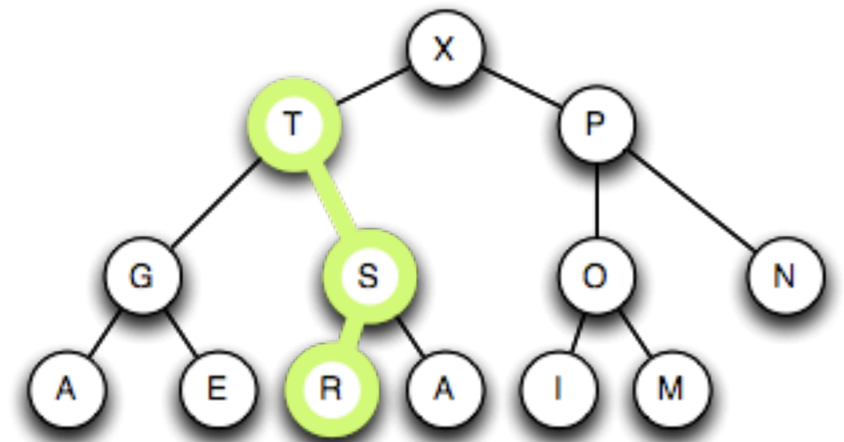


Promoción de **un** nodo (recorrido hacia arriba)

- un nodo tiene un valor de llave más grande que su padre:
  - intercambiar al nodo con su padre.
  - repetir el proceso hasta verificar orden.

```
void fixUp( Item a[], int k )
{
    while ( k>1 && a[k/2]<a[k] )
    {
        exch(a[k], a[k/2]);
        k = k/2;
    }
}
```





Degradación de **un** nodo (recorrido hacia abajo)

Degradación de **un** nodo (recorrido hacia abajo)

- un nodo tiene un valor de llave más pequeño que uno o sus dos hijos:

Degradación de **un** nodo (recorrido hacia abajo)

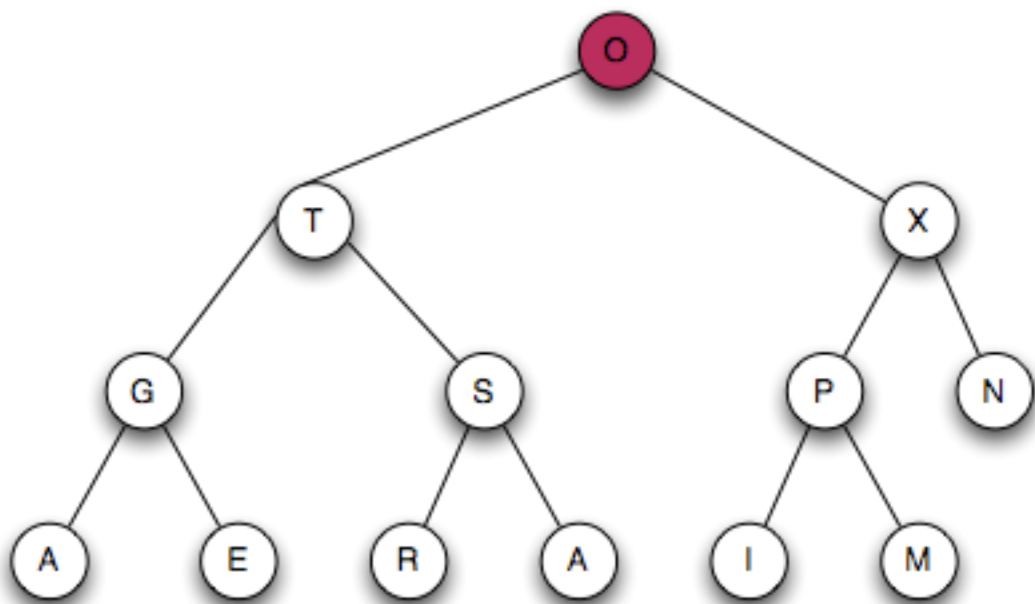
- un nodo tiene un valor de llave más pequeño que uno o sus dos hijos:
  - intercambiar al nodo con su hijo mayor.

Degradación de **un** nodo (recorrido hacia abajo)

- un nodo tiene un valor de llave más pequeño que uno o sus dos hijos:
  - intercambiar al nodo con su hijo mayor.
  - repetir el proceso hasta verificar orden o llegar al final.

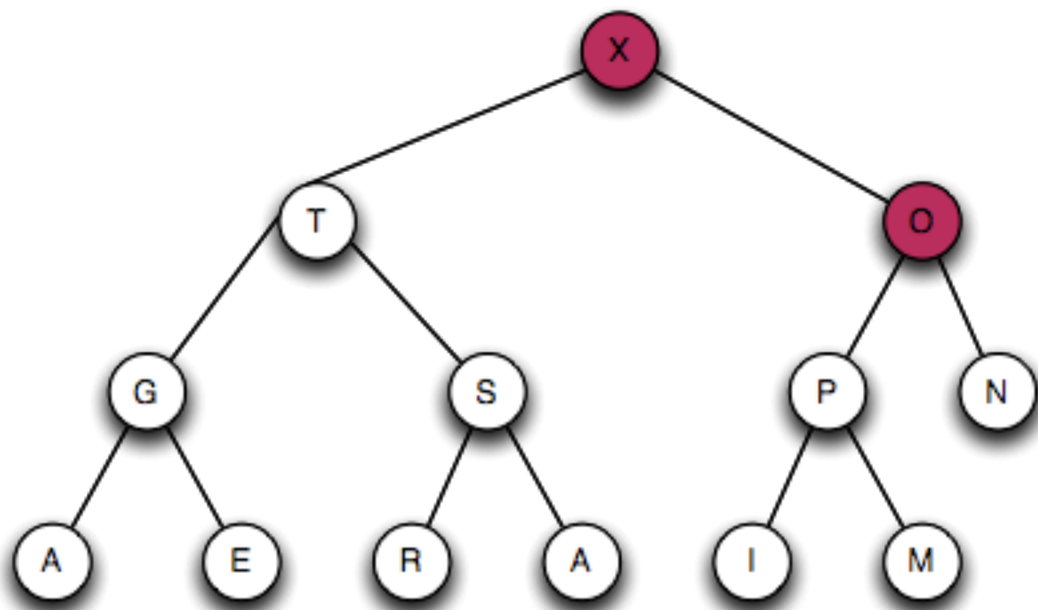
Degradación de **un** nodo (recorrido hacia abajo)

- un nodo tiene un valor de llave más pequeño que uno o sus dos hijos:
  - intercambiar al nodo con su hijo mayor.
  - repetir el proceso hasta verificar orden o llegar al final.



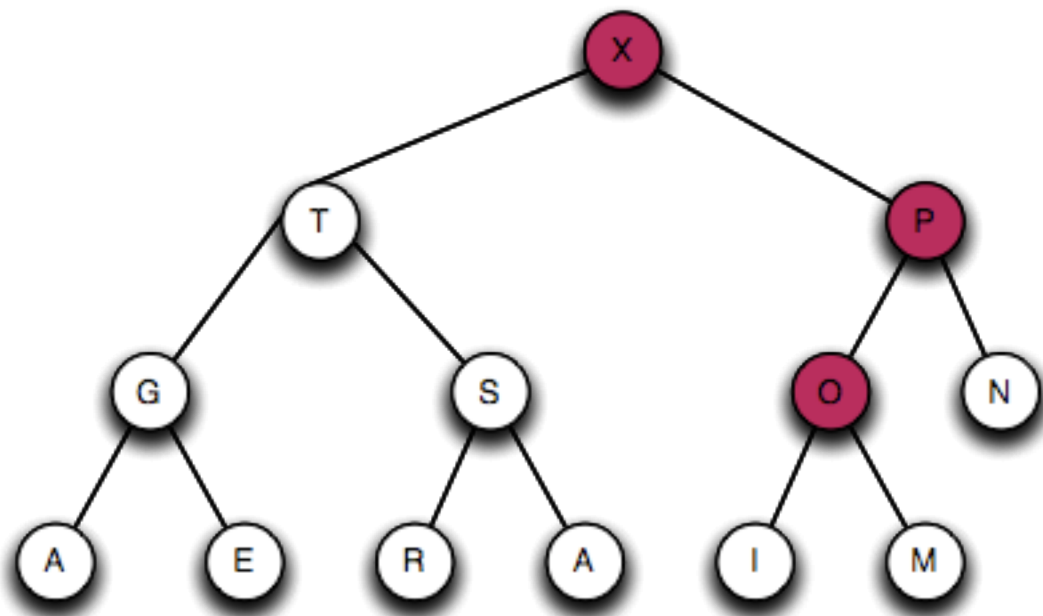
Degradación de **un** nodo (recorrido hacia abajo)

- un nodo tiene un valor de llave más pequeño que uno o sus dos hijos:
  - intercambiar al nodo con su hijo mayor.
  - repetir el proceso hasta verificar orden o llegar al final.



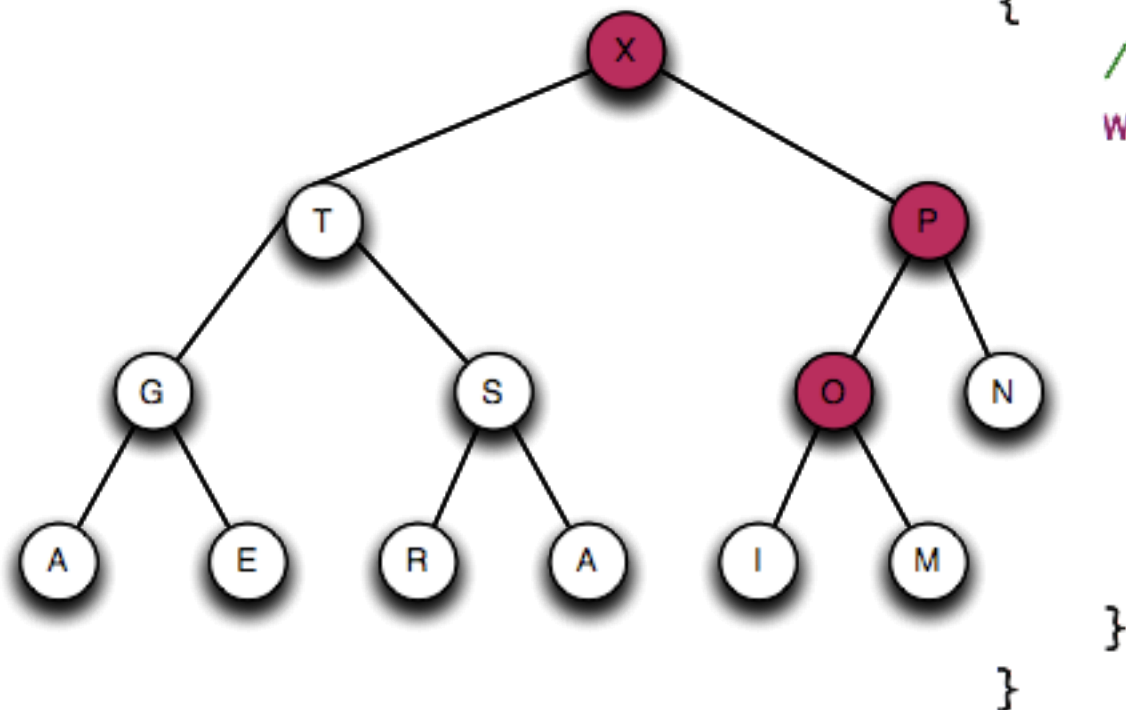
Degradación de **un** nodo (recorrido hacia abajo)

- un nodo tiene un valor de llave más pequeño que uno o sus dos hijos:
  - intercambiar al nodo con su hijo mayor.
  - repetir el proceso hasta verificar orden o llegar al final.



Degradación de **un** nodo (recorrido hacia abajo)

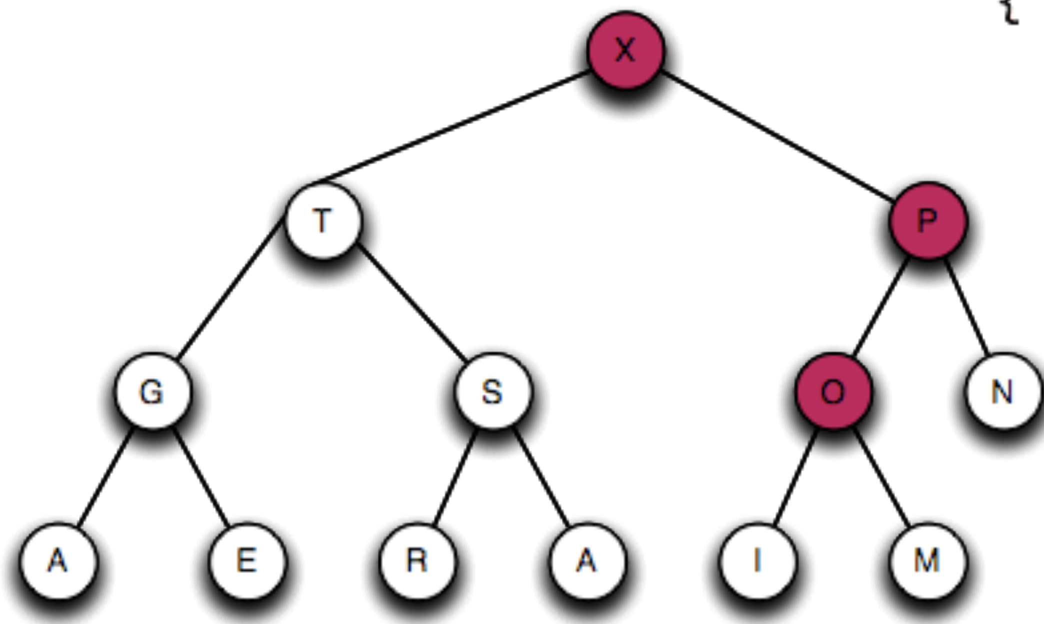
- un nodo tiene un valor de llave más pequeño que uno o sus dos hijos:
  - intercambiar al nodo con su hijo mayor.
  - repetir el proceso hasta verificar orden o llegar al final.




```
void fixDown( Item a[], int k, int N )
{
    // los hijos del nodo k estan en las posiciones 2k y 2k+1
    while( 2*k <= N ) {
        int j = 2*k;
        if( j < N && a[j] < a[j+1] )
            j++;
        if( !(a[k] < a[j] ))
            break;
        exch( a[k], a[j] );
        k = j;
    }
}
```

Degradación de **un** nodo (recorrido hacia abajo)

- un nodo tiene un valor de llave más pequeño que uno o sus dos hijos:
    - intercambiar al nodo con su hijo mayor.
    - repetir el proceso hasta verificar orden o llegar al final.




```
void fixDown( Item a[], int k, int N )
{
    // los hijos del nodo k estan en las posiciones 2k y 2k+1
    while( 2*k <= N ) {
        int j = 2*k;
        if( j < N && a[j] < a[j+1] )
            j++;
        if( !(a[k] < a[j] ))
            break;
        exch( a[k], a[j] );
        k = j;
    }
}
```

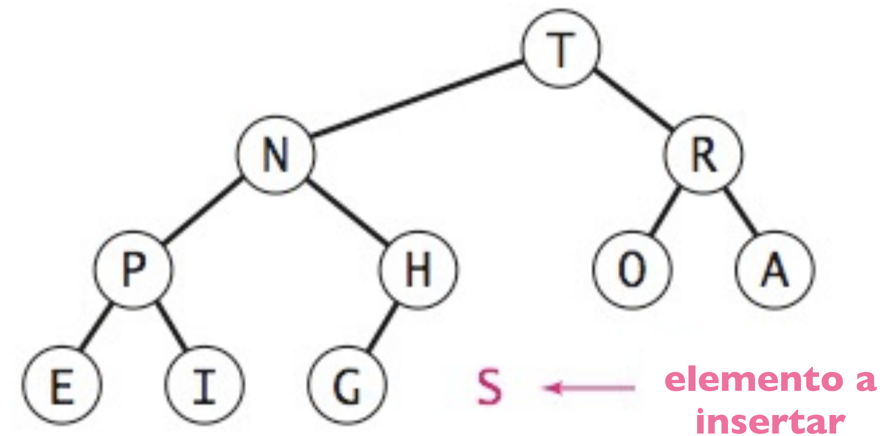
- Se necesita conocer el tamaño ***N*** del montículo.

Operaciones en colas de prioridad con montículos: **Inserción** de un nodo

- Agregar un nodo al final, luego promoverlo.

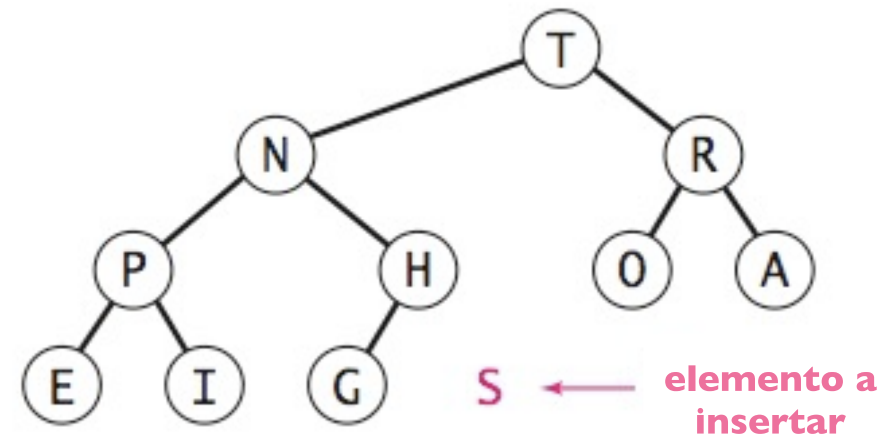
Operaciones en colas de prioridad con montículos: **Inserción** de un nodo

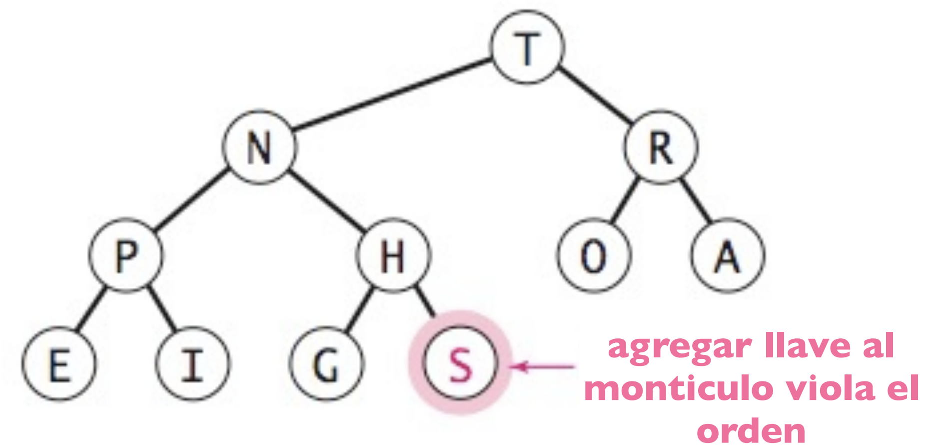
- Agregar un nodo al final, luego promoverlo.



Operaciones en colas de prioridad con montículos: **Inserción** de un nodo

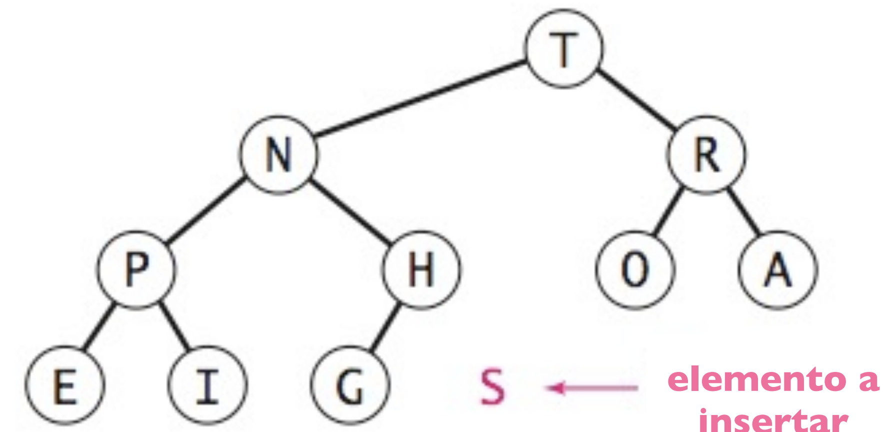
- Agregar un nodo al final, luego promoverlo.

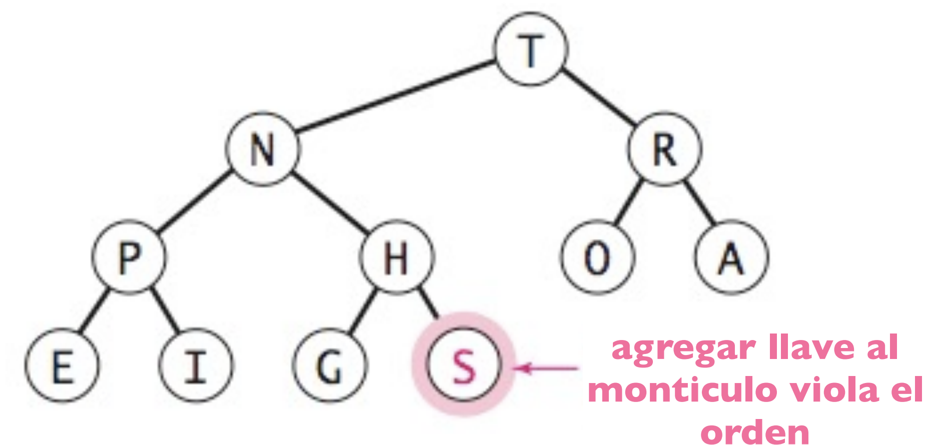


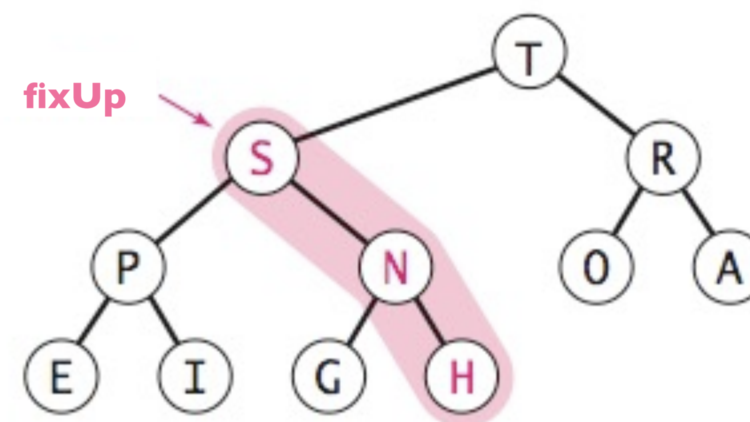


Operaciones en colas de prioridad con montículos: **Inserción** de un nodo

- Agregar un nodo al final, luego promoverlo.

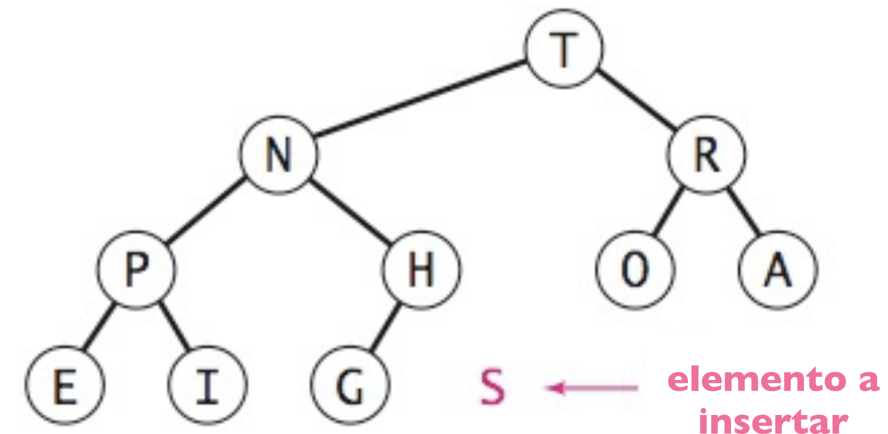




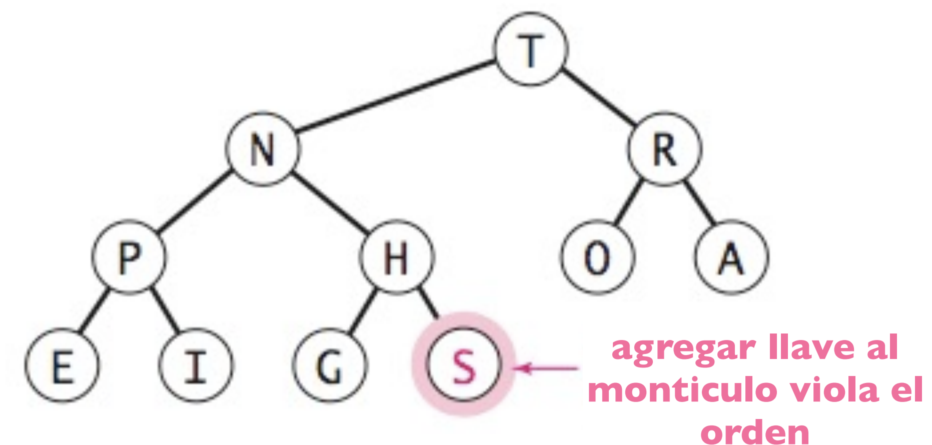


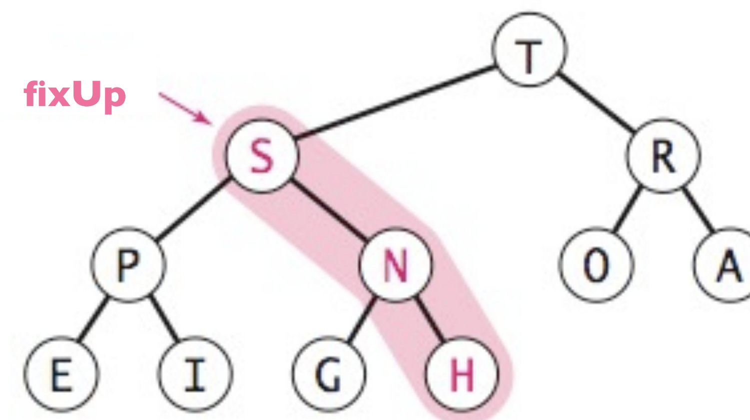
Operaciones en colas de prioridad con montículos: **Inserción** de un nodo

- Agregar un nodo al final, luego promoverlo.




```
void insert( Item x )
{
     pq[++N] = x;
     fixUp(a,N);
}
```



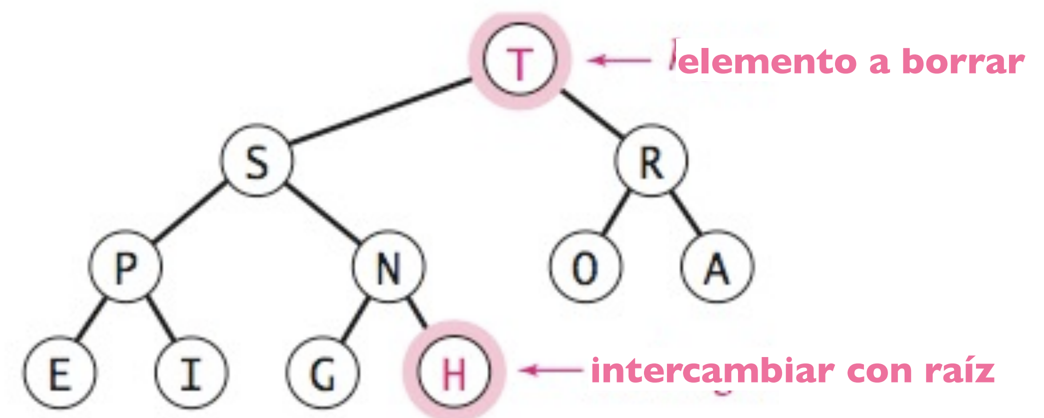


Operaciones en colas de prioridad con montículos: **Eliminación** de el nodo mayor

- Intercambiar la raíz con el último nodo, luego degradarlo.

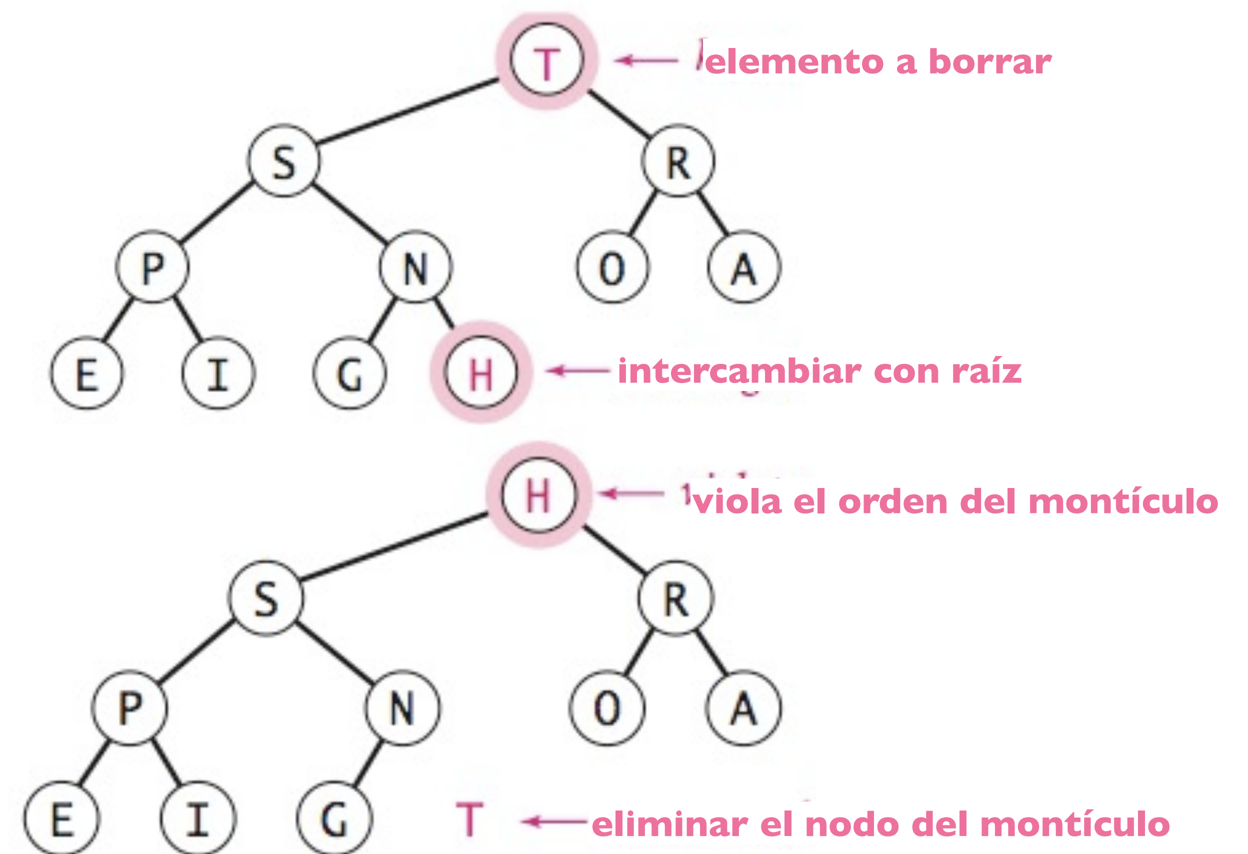
Operaciones en colas de prioridad con montículos: **Eliminación** de el nodo mayor

- Intercambiar la raíz con el último nodo, luego degradarlo.



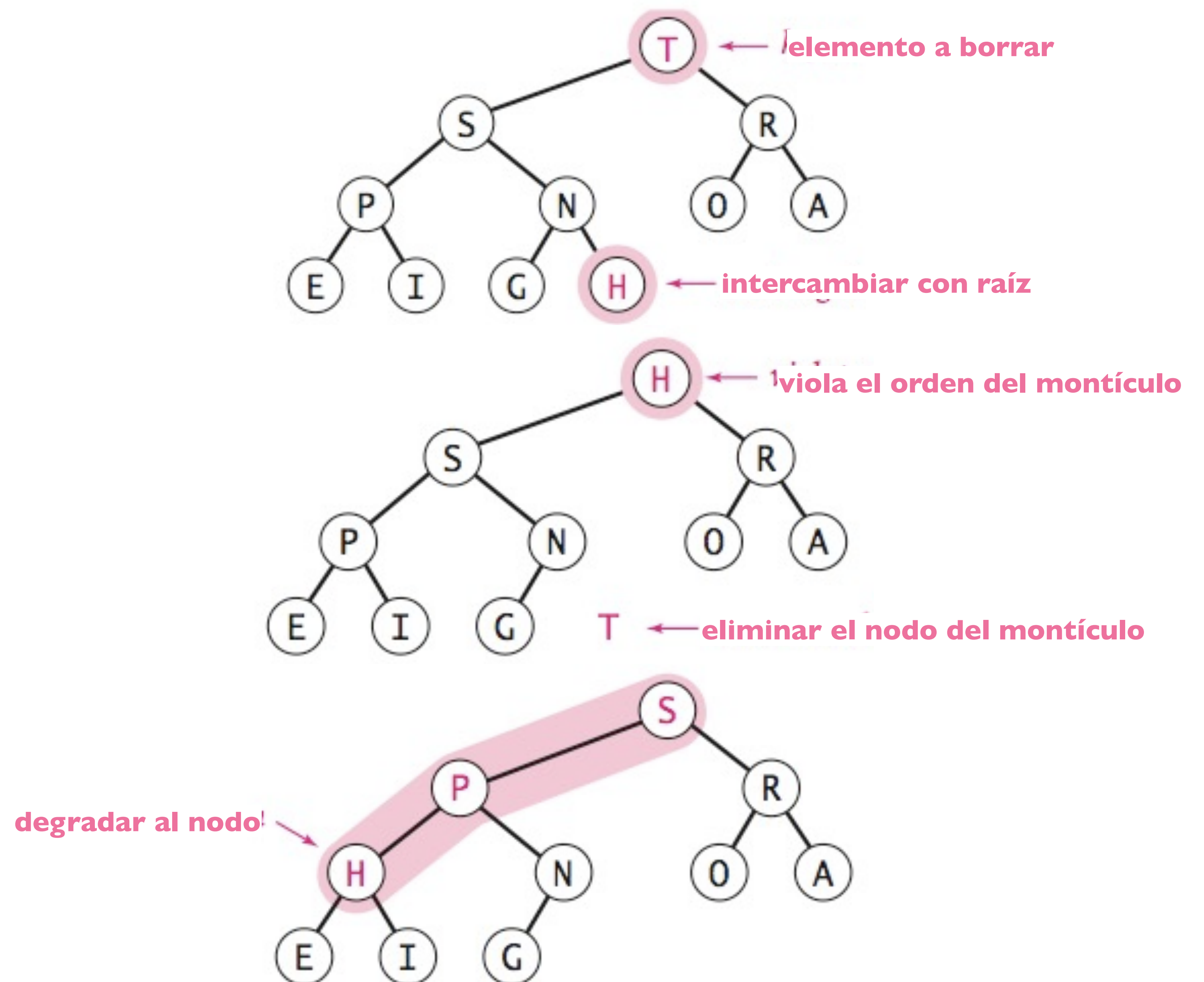
Operaciones en colas de prioridad con montículos: **Eliminación** de el nodo mayor

- Intercambiar la raíz con el último nodo, luego degradarlo.



Operaciones en colas de prioridad con montículos: **Eliminación** de el nodo mayor

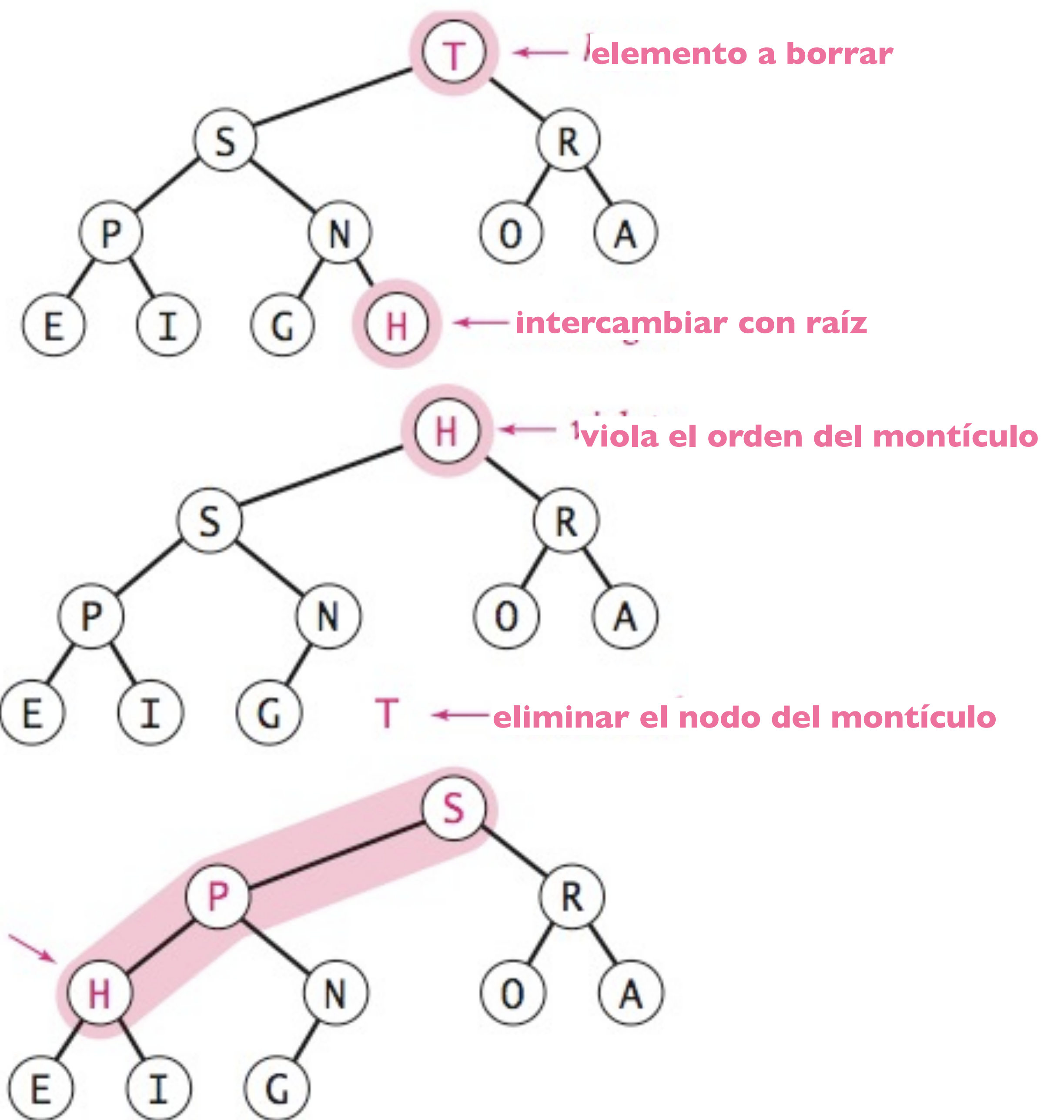
- Intercambiar la raíz con el último nodo, luego degradarlo.



Operaciones en colas de prioridad con montículos: **Eliminación** de el nodo mayor

- Intercambiar la raíz con el último nodo, luego degradarlo.

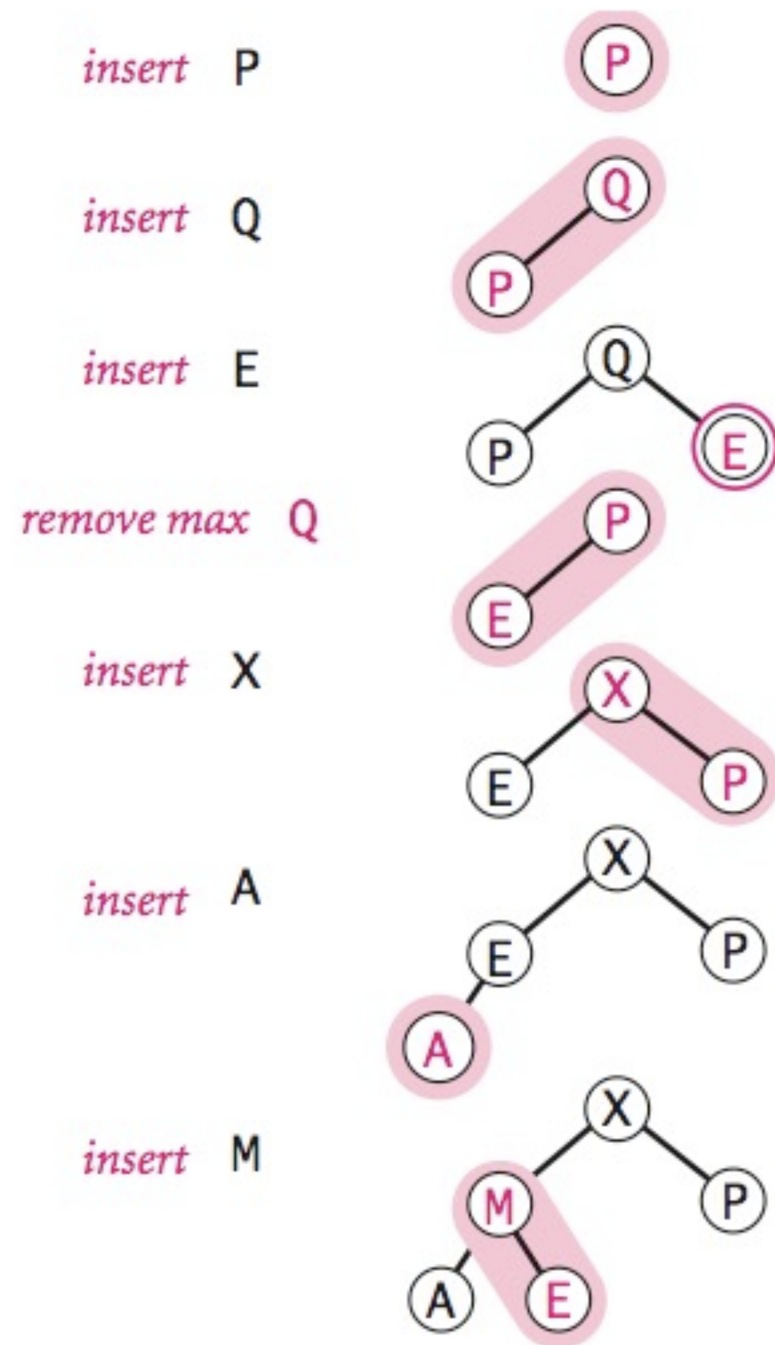
```
Item delMax()
{
    Item max = pq[1];
    exch( pq[1], pq[N] );
    fixDown(pq, 1, N-1);
    N--;
    return max;
}
```



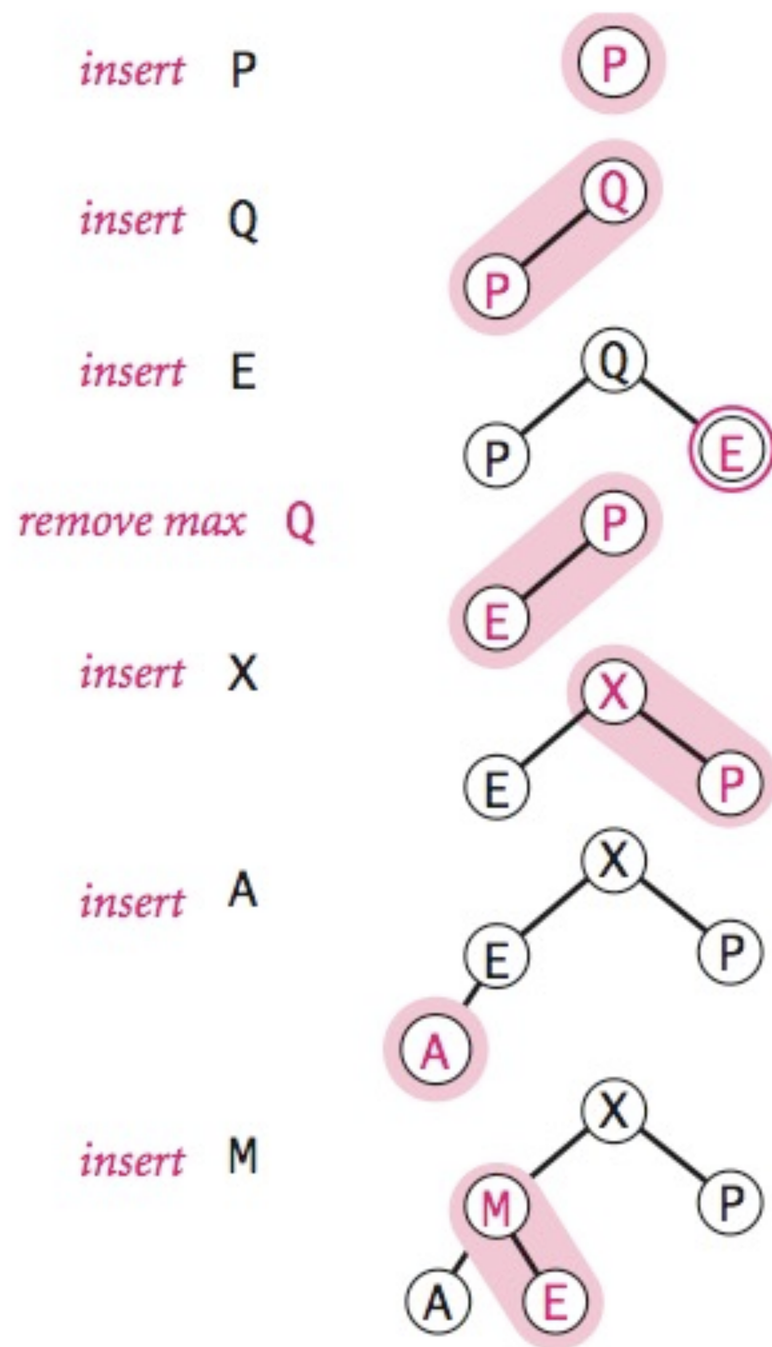
Ejemplo de implementación de una clase cola de prioridad con montículos

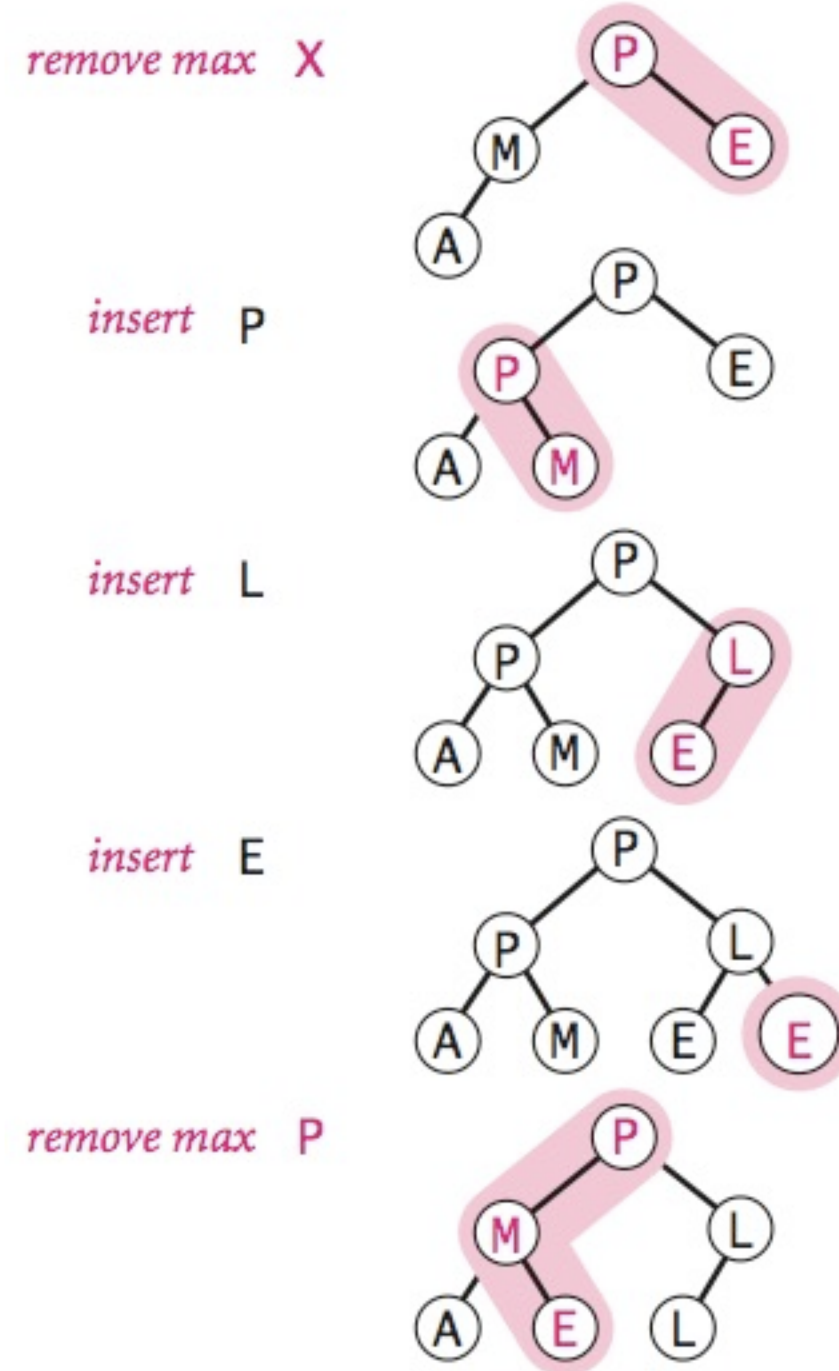
```
class PQ
{
    private:

        Item *pq;
        int N;

    public:

        PQ(int maxN) { // constructor
            pq = new Item[maxN+1];
            N = 0;
        }

        int empty() {
            return N == 0;
        }

        void insert(Item item) {
            pq[++N] = item;
            fixUp(pq, N);
        }

        Item delMax() {
            Item max = pq[1];
            exch( pq[1], pq[N] );
            fixDown( pq, 1, N-1 );
            return pq[N--];
        }
};
```

Operaciones en colas de prioridad con montículos



Operaciones en colas de prioridad con montículos





Operaciones en colas de prioridad con montículos

Operaciones en colas de prioridad con montículos

- Las operaciones insert y delMax para el ADT de una cola de prioridad de tamaño ***N*** se pueden implementar con montículos de tal manera que:

Operaciones en colas de prioridad con montículos

- Las operaciones insert y delMax para el ADT de una cola de prioridad de tamaño ***N*** se pueden implementar con montículos de tal manera que:

Operaciones en colas de prioridad con montículos

- Las operaciones insert y delMax para el ADT de una cola de prioridad de tamaño ***N*** se pueden implementar con montículos de tal manera que:
  - insert no tome más de ***lg N*** comparaciones

Operaciones en colas de prioridad con montículos

- Las operaciones insert y delMax para el ADT de una cola de prioridad de tamaño ***N*** se pueden implementar con montículos de tal manera que:
  - insert no tome más de ***lg N*** comparaciones
  - delMax no tome más de ***2 lg N*** comparaciones.

Operaciones en colas de prioridad con montículos

- Las operaciones insert y delMax para el ADT de una cola de prioridad de tamaño ***N*** se pueden implementar con montículos de tal manera que:
  - insert no tome más de ***lg N*** comparaciones
  - delMax no tome más de ***2 lg N*** comparaciones.
    - comparación para encontrar cuál es el hijo más grande.

Operaciones en colas de prioridad con montículos

- Las operaciones insert y delMax para el ADT de una cola de prioridad de tamaño ***N*** se pueden implementar con montículos de tal manera que:
  - insert no tome más de ***lg N*** comparaciones
  - delMax no tome más de ***2 lg N*** comparaciones.
    - comparación para encontrar cuál es el hijo más grande.
    - comparación para saber si el hijo debe ser promovido.

Operaciones en colas de prioridad con montículos

- Las operaciones insert y delMax para el ADT de una cola de prioridad de tamaño ***N*** se pueden implementar con montículos de tal manera que:
  - insert no tome más de ***lg N*** comparaciones
  - delMax no tome más de ***2 lg N*** comparaciones.
    - comparación para encontrar cuál es el hijo más grande.
    - comparación para saber si el hijo debe ser promovido.

Operaciones en colas de prioridad con montículos

- Las operaciones insert y delMax para el ADT de una cola de prioridad de tamaño ***N*** se pueden implementar con montículos de tal manera que:
  - insert no tome más de ***lg N*** comparaciones
  - delMax no tome más de ***2 lg N*** comparaciones.
    - comparación para encontrar cuál es el hijo más grande.
    - comparación para saber si el hijo debe ser promovido.
- La construcción de un montículo toma un tiempo proporcional a ***N lg N*** en el peor caso.

Operaciones en colas de prioridad con montículos

Operaciones en colas de prioridad con montículos

- Las operaciones básicas fixUp y fixDown nos permiten también una implementación directa de funciones que:

Operaciones en colas de prioridad con montículos

- Las operaciones básicas fixUp y fixDown nos permiten también una implementación directa de funciones que:
  - cambian la prioridad de un nodo cualquiera,

Operaciones en colas de prioridad con montículos

- Las operaciones básicas fixUp y fixDown nos permiten también una implementación directa de funciones que:
  - cambian la prioridad de un nodo cualquiera,
  - eliminan un nodo cualquiera.

Operaciones en colas de prioridad con montículos

- Las operaciones básicas fixUp y fixDown nos permiten también una implementación directa de funciones que:
  - cambian la prioridad de un nodo cualquiera,
  - eliminan un nodo cualquiera.
- Estas operaciones se pueden implementar con montículos con ***2 lg N*** comparaciones para una cola de prioridad de tamaño ***N***.

Operaciones en colas de prioridad con montículos

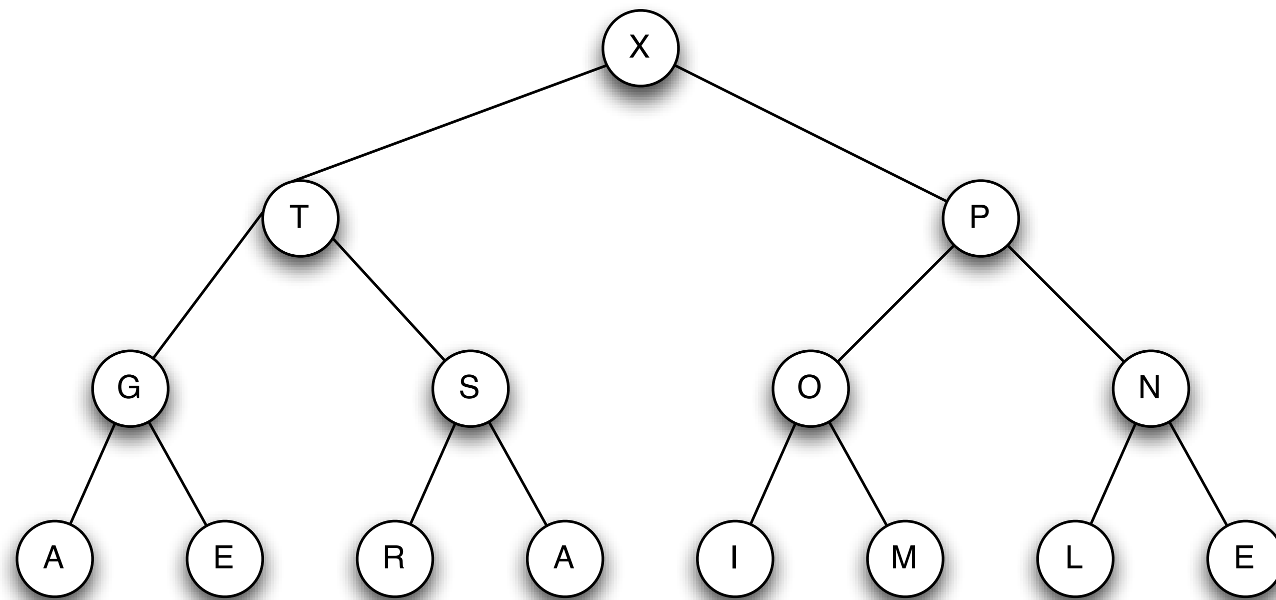
- Las operaciones básicas fixUp y fixDown nos permiten también una implementación directa de funciones que:
  - cambian la prioridad de un nodo cualquiera,
  - eliminan un nodo cualquiera.
- Estas operaciones se pueden implementar con montículos con ***2 lg N*** comparaciones para una cola de prioridad de tamaño ***N***.
- La operación join (eficiente) requiere una estructura de datos más sofisticada.

Operaciones en colas de prioridad con montículos

- Las operaciones básicas fixUp y fixDown nos permiten también una implementación directa de funciones que:
  - cambian la prioridad de un nodo cualquiera,
  - eliminan un nodo cualquiera.
- Estas operaciones se pueden implementar con montículos con ***2 lg N*** comparaciones para una cola de prioridad de tamaño ***N***.
- La operación join (eficiente) requiere una estructura de datos más sofisticada.
- Los montículos son suficientes para implementaciones eficientes de colas de prioridad (excepto operaciones *join* frecuentes y largas).

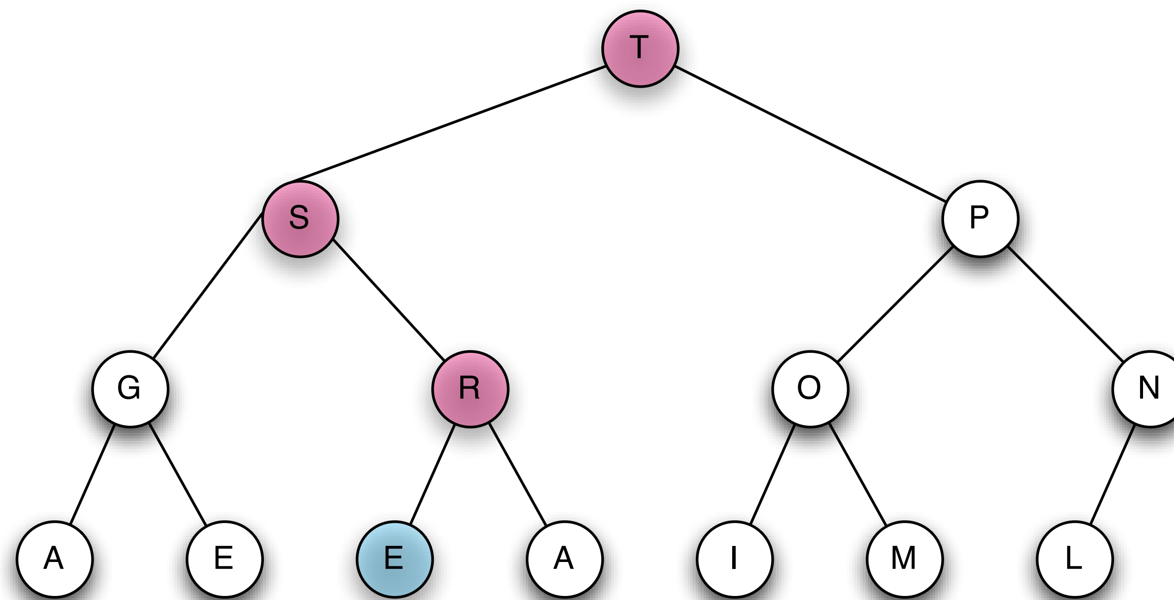
Algoritmos de ordenamiento con colas de prioridad

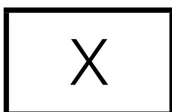
- Insertar todas las llaves a ser ordenadas en la cola de prioridad.
- Usar la operación delMax para obtener las llaves en orden decreciente.



Algoritmos de ordenamiento con colas de prioridad

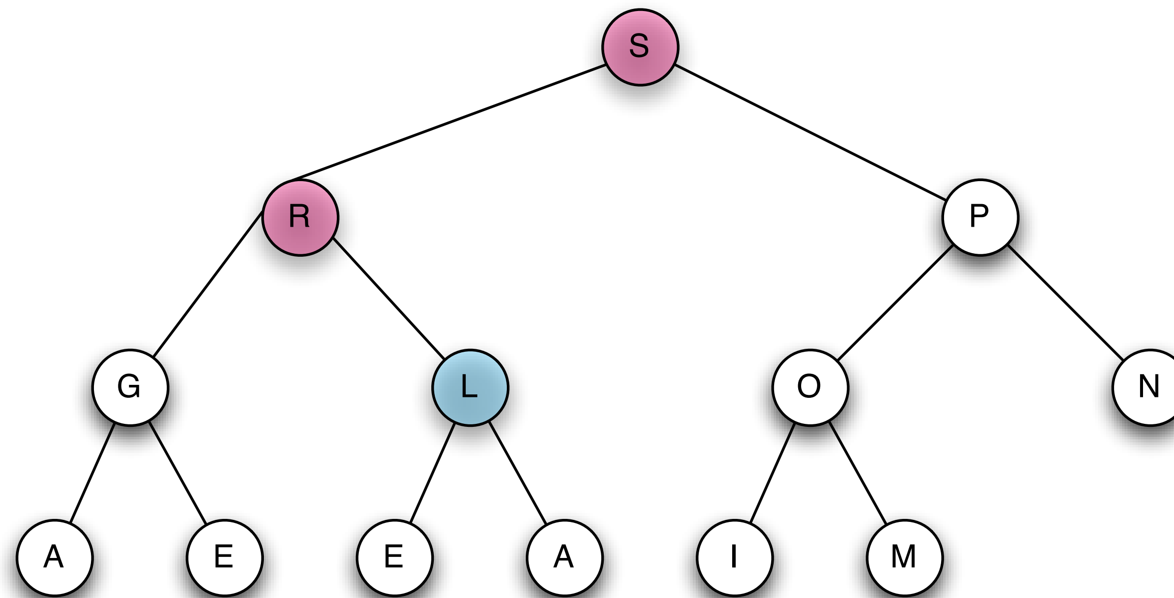
- Insertar todas las llaves a ser ordenadas en la cola de prioridad.
- Usar la operación delMax para obtener las llaves en orden decreciente.





Algoritmos de ordenamiento con colas de prioridad

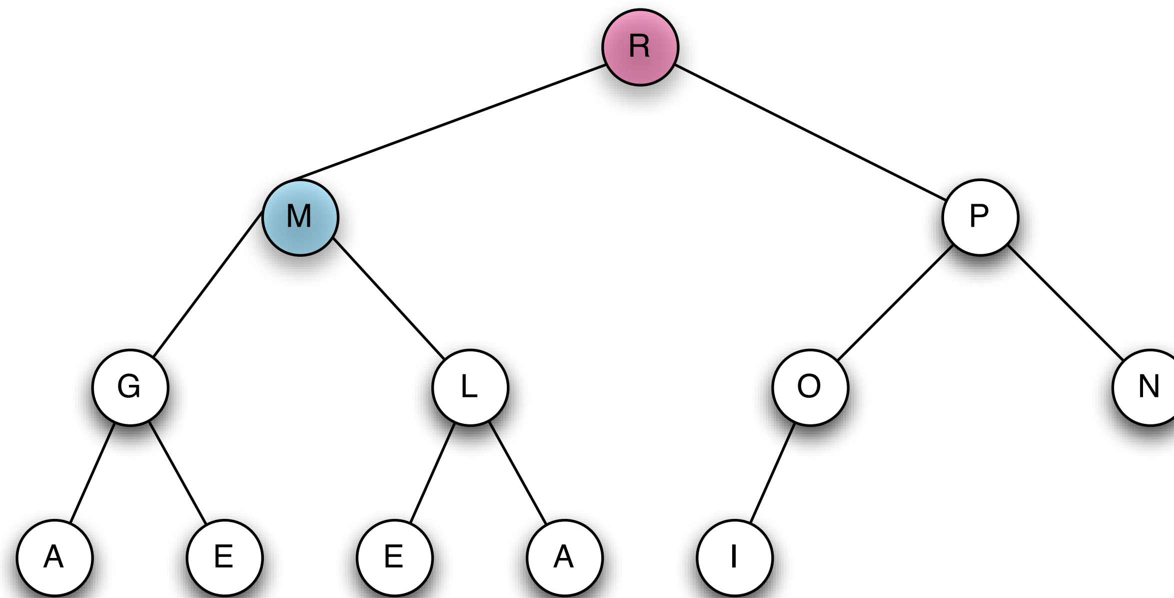
- Insertar todas las llaves a ser ordenadas en la cola de prioridad.
- Usar la operación delMax para obtener las llaves en orden decreciente.



X	T

Algoritmos de ordenamiento con colas de prioridad

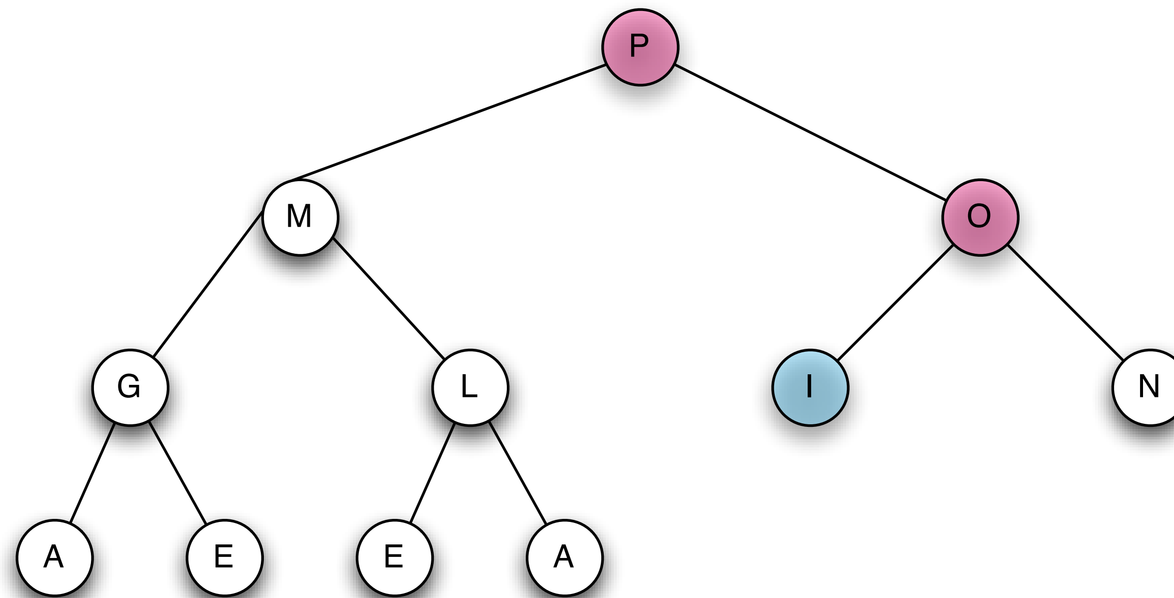
- Insertar todas las llaves a ser ordenadas en la cola de prioridad.
- Usar la operación delMax para obtener las llaves en orden decreciente.

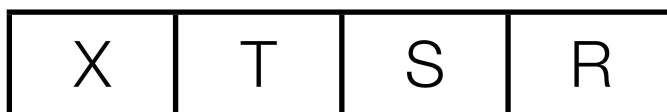


X	T	S

Algoritmos de ordenamiento con colas de prioridad

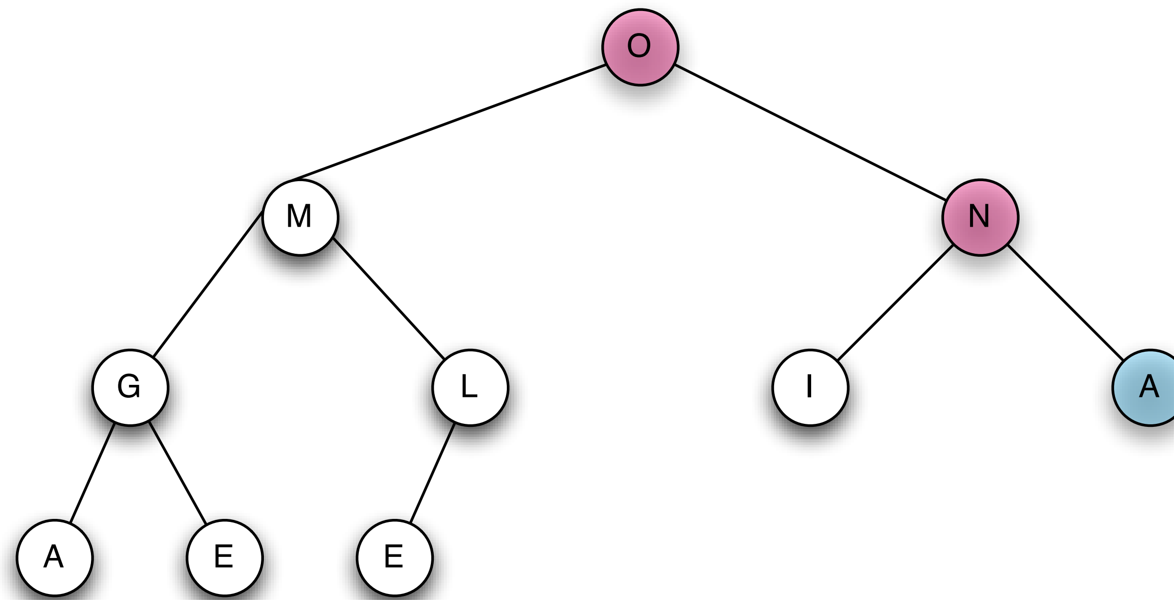
- Insertar todas las llaves a ser ordenadas en la cola de prioridad.
- Usar la operación delMax para obtener las llaves en orden decreciente.

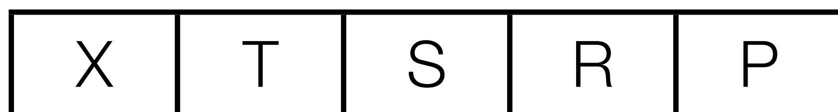




Algoritmos de ordenamiento con colas de prioridad

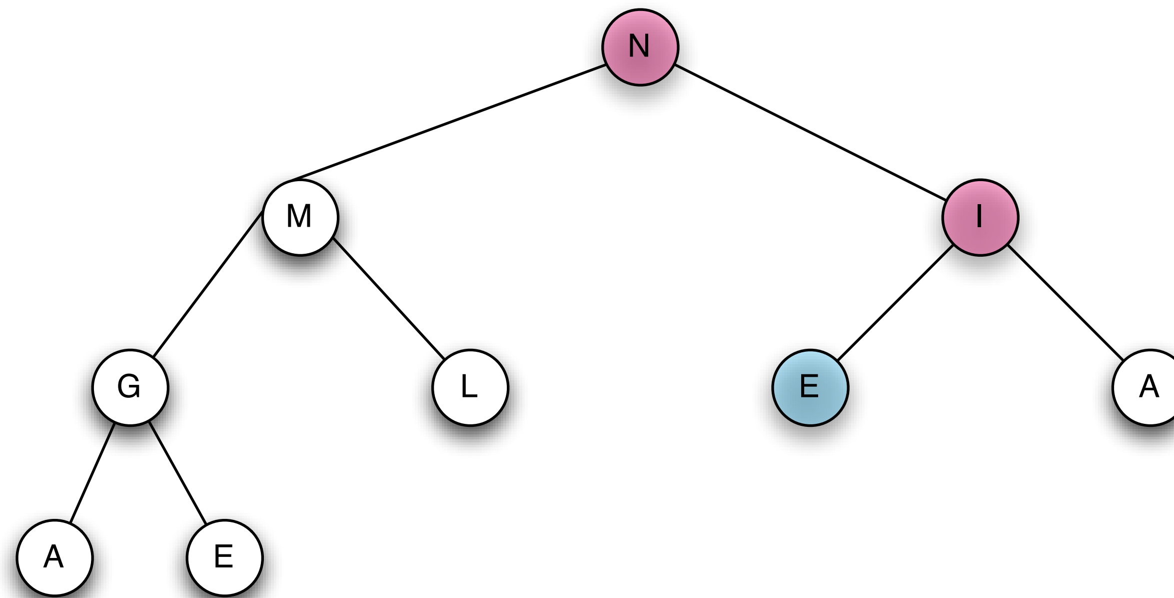
- Insertar todas las llaves a ser ordenadas en la cola de prioridad.
- Usar la operación delMax para obtener las llaves en orden decreciente.





Algoritmos de ordenamiento con colas de prioridad

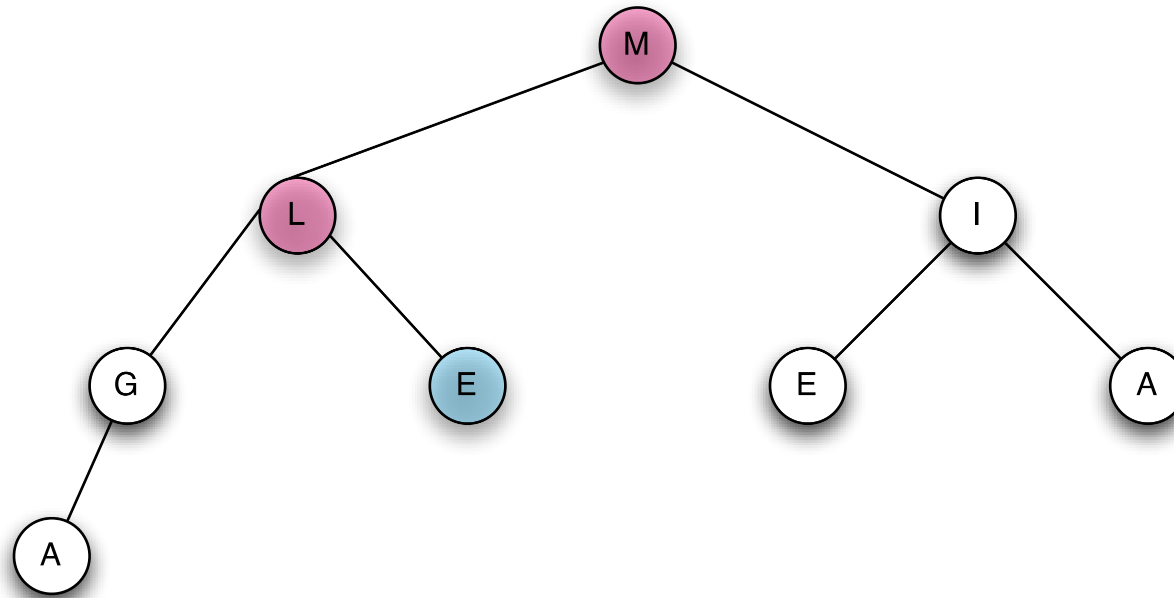
- **Insertar** todas las **llaves** a ser ordenadas en la **cola de prioridad**.
- Usar la operación **delMax** para obtener las llaves en **orden decreciente**.



X	T	S	R	P	O

Algoritmos de ordenamiento con colas de prioridad

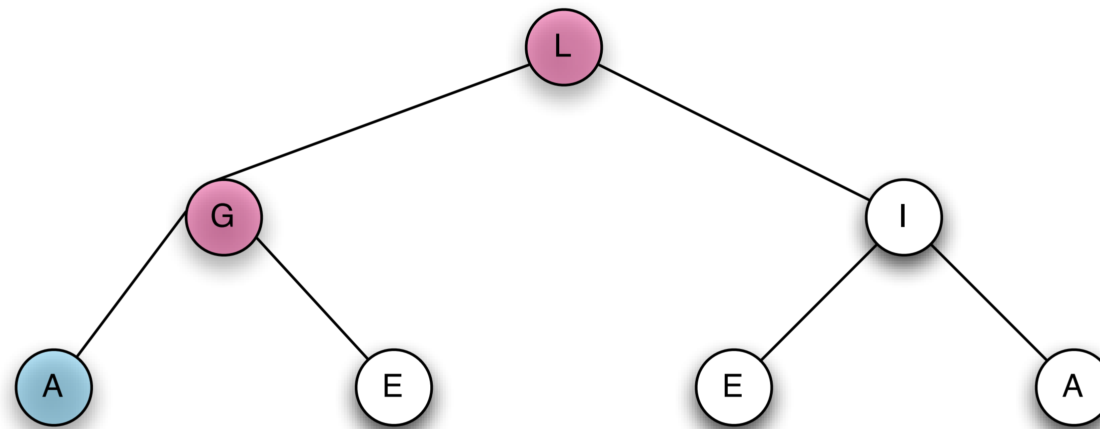
- Insertar todas las llaves a ser ordenadas en la cola de prioridad.
- Usar la operación delMax para obtener las llaves en orden decreciente.



X	T	S	R	P	O	N

Algoritmos de ordenamiento con colas de prioridad

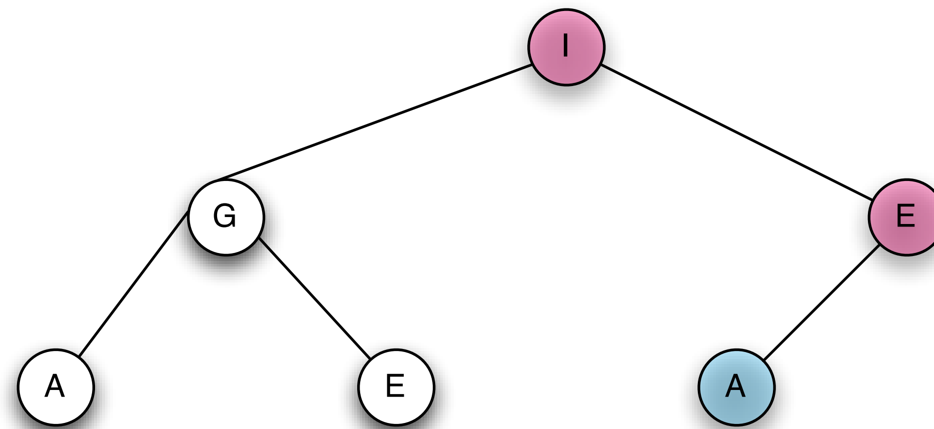
- Insertar todas las llaves a ser ordenadas en la cola de prioridad.
- Usar la operación delMax para obtener las llaves en orden decreciente.



X	T	S	R	P	O	N	M

Algoritmos de ordenamiento con colas de prioridad

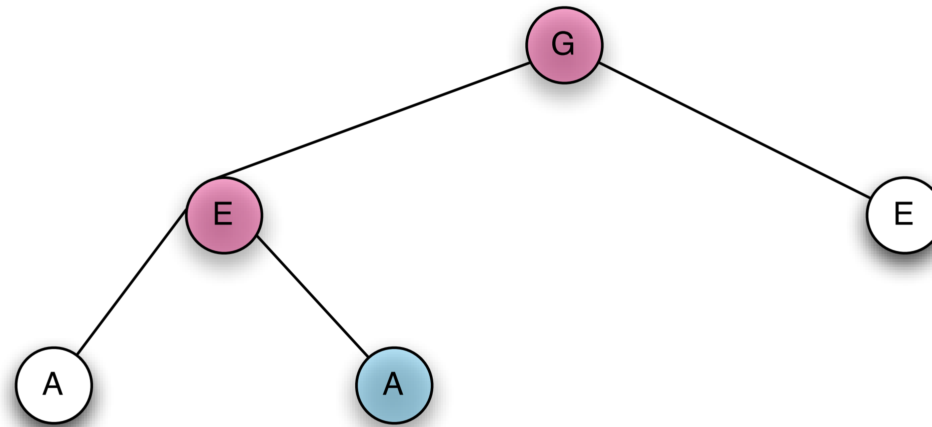
- Insertar todas las llaves a ser ordenadas en la cola de prioridad.
- Usar la operación delMax para obtener las llaves en orden decreciente.



X	T	S	R	P	O	N	M	L

Algoritmos de ordenamiento con colas de prioridad

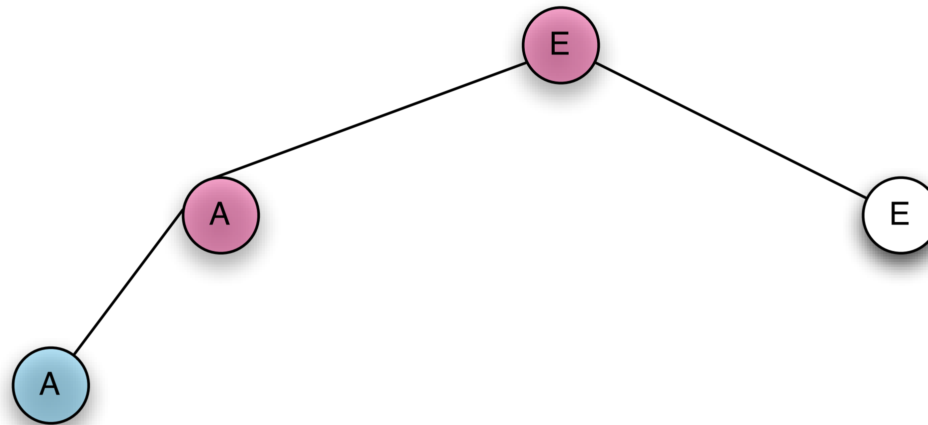
- Insertar todas las llaves a ser ordenadas en la cola de prioridad.
- Usar la operación delMax para obtener las llaves en orden decreciente.



X	T	S	R	P	O	N	M	L	I

Algoritmos de ordenamiento con colas de prioridad

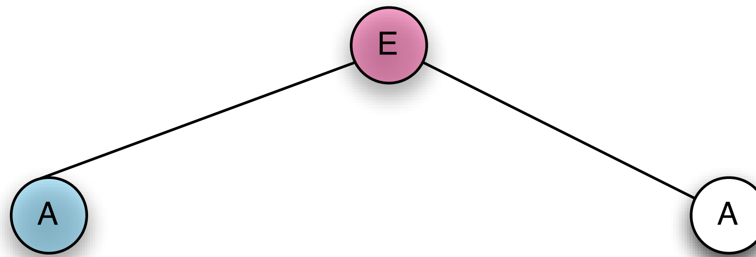
- Insertar todas las llaves a ser ordenadas en la cola de prioridad.
- Usar la operación delMax para obtener las llaves en orden decreciente.



X	T	S	R	P	O	N	M	L	I	G

Algoritmos de ordenamiento con colas de prioridad

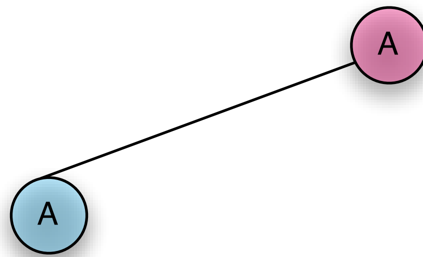
- Insertar todas las llaves a ser ordenadas en la cola de prioridad.
- Usar la operación delMax para obtener las llaves en orden decreciente.



X	T	S	R	P	O	N	M	L	I	G	E

Algoritmos de ordenamiento con colas de prioridad

- Insertar todas las llaves a ser ordenadas en la cola de prioridad.
- Usar la operación delMax para obtener las llaves en orden decreciente.



X	T	S	R	P	O	N	M	L	I	G	E	E

Algoritmos de ordenamiento con colas de prioridad

- Insertar todas las llaves a ser ordenadas en la cola de prioridad.
- Usar la operación delMax para obtener las llaves en orden decreciente.

A

X	T	S	R	P	O	N	M	L	I	G	E	E	A	A

Algoritmos de ordenamiento con colas de prioridad

```
void PQsort( Item a[], int l, int r )
{
    int k;
    pq(r-l+1);
    for ( k=l; k<=r; k++ )
        pq.insert(a[k]);
    for ( k=r; k>=l; k-- )
        a[k] = pq.delMax();
}
```

Algoritmos de ordenamiento con colas de prioridad

- Es un método comparativamente **ineficiente** porque hace **copias** de los **elementos** a ordenar.

```
void PQsort( Item a[], int l, int r )
{
    int k;
    pq(r-l+1);
    for ( k=l; k<=r; k++ )
        pq.insert(a[k]);
    for ( k=r; k>=l; k-- )
        a[k] = pq.delMax();
}
```

Algoritmos de ordenamiento con colas de prioridad

- Es un método comparativamente ineficiente porque hace copias de los elementos a ordenar.
- Usar ***N*** inserciones sucesivas no es la mejor manera de crear un montículo con los ***N*** elementos dados.

```
void PQsort( Item a[], int l, int r )
{
    int k;
    pq(r-l+1);
    for ( k=l; k<=r; k++ )
        pq.insert(a[k]);
    for ( k=r; k>=l; k-- )
        a[k] = pq.delMax();
}
```

Heapsort

Heapsort

- Adaptar la idea básica del ordenamiento con colas de prioridad sin necesidad de espacio extra.

Heapsort

- Adaptar la idea básica del ordenamiento con colas de prioridad sin necesidad de espacio extra.
    - Mantener el montículo en el arreglo a ser ordenado.

Heapsort

- Adaptar la idea básica del ordenamiento con colas de prioridad sin necesidad de espacio extra.
    - Mantener el montículo en el arreglo a ser ordenado.
- En lugar de construir el montículo por medio de inserciones sucesivas hacemos sub-montículos de abajo hacia arriba (derecha a izquierda).

Heapsort

- Adaptar la idea básica del ordenamiento con colas de prioridad sin necesidad de espacio extra.
    - Mantener el montículo en el arreglo a ser ordenado.
- En lugar de construir el montículo por medio de inserciones sucesivas hacemos sub-montículos de abajo hacia arriba (derecha a izquierda).
- Idea básica: insertar los elementos en un montículo, extraer los mayores en el montículo mismo.

Heapsort

Heapsort

- Método basado en comparaciones.

Heapsort

- Método basado en comparaciones.
- Parte de la familia de tipo selection sort.

Heapsort

- Método basado en comparaciones.
- Parte de la familia de tipo selection sort.
- Complejidad en el peor caso de $O(n \log n)$.

Heapsort

- Método basado en comparaciones.
- Parte de la familia de tipo selection sort.
- Complejidad en el peor caso de $O(n \log n)$.
- Algoritmo in-place (no requiere memoria extra).

Heapsort

- Método basado en comparaciones.
- Parte de la familia de tipo selection sort.
- Complejidad en el peor caso de $O(n \log n)$.
- Algoritmo in-place (no requiere memoria extra).
- Algoritmo no estable.

Heapsort

- **Primer paso:** Construir un montículo utilizando un método de abajo hacia arriba.

Heapsort

- Primer paso: Construir un montículo utilizando un método de abajo hacia arriba.

```
for( int k=N/2; k>=1; k-- )
    fixDown( pq, k, N );
```

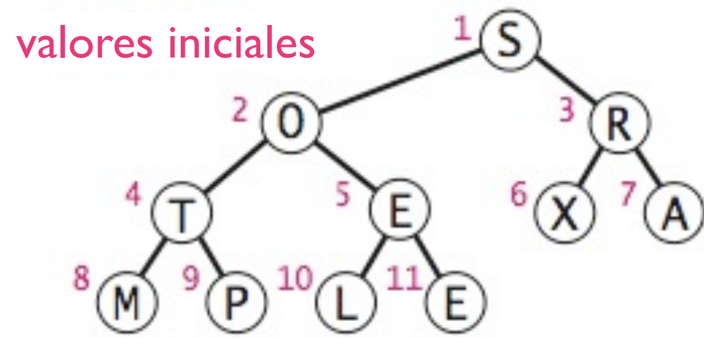
Heapsort

- Primer paso: Construir un montículo utilizando un método de abajo hacia arriba.

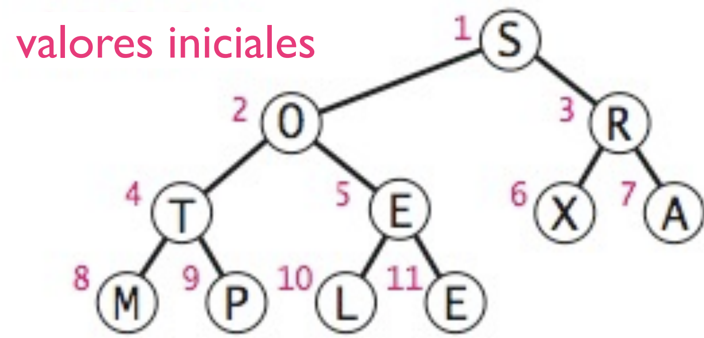
```
for( int k=N/2; k>=1; k-- )
    fixDown( pq, k, N );
```

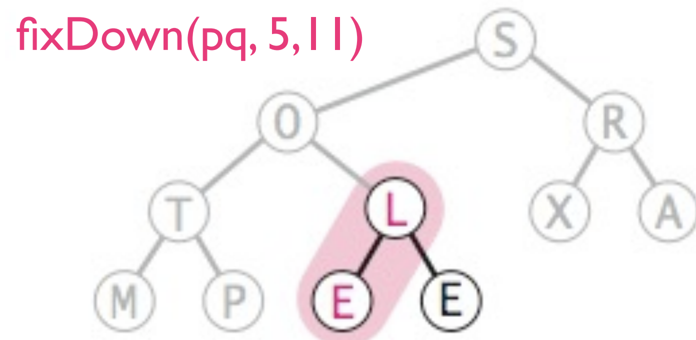
- No necesitamos tomar en cuenta los sub-montículos de tamaño 1.

Heapsort - Creación de montículo de abajo hacia arriba

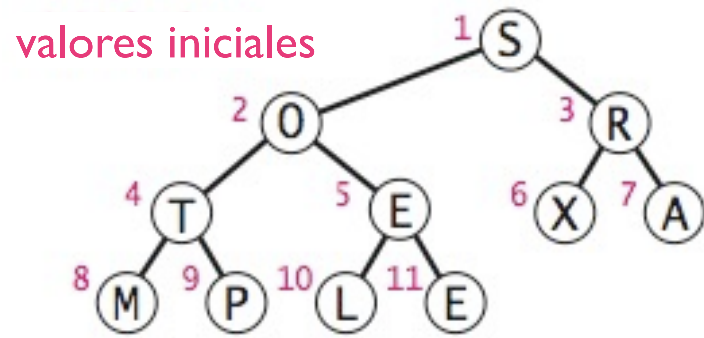


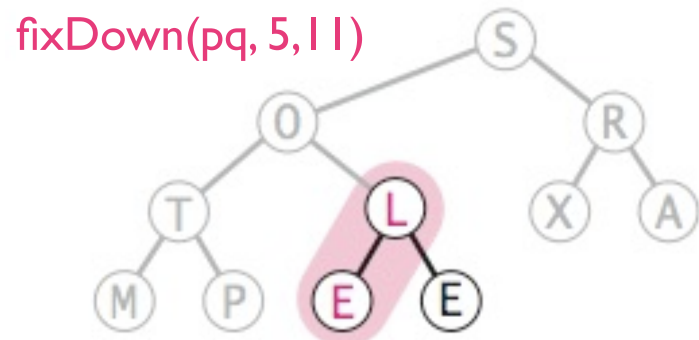
Heapsort - Creación de montículo de abajo hacia arriba

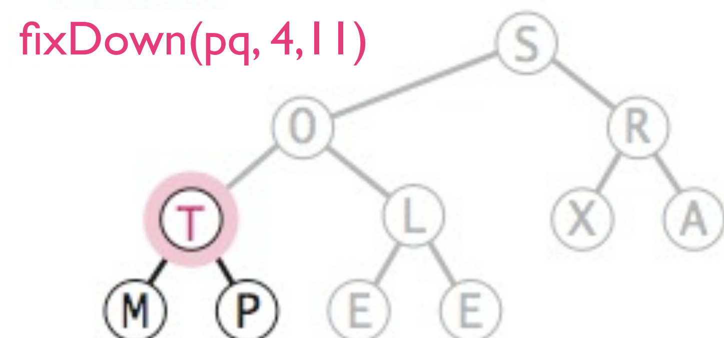




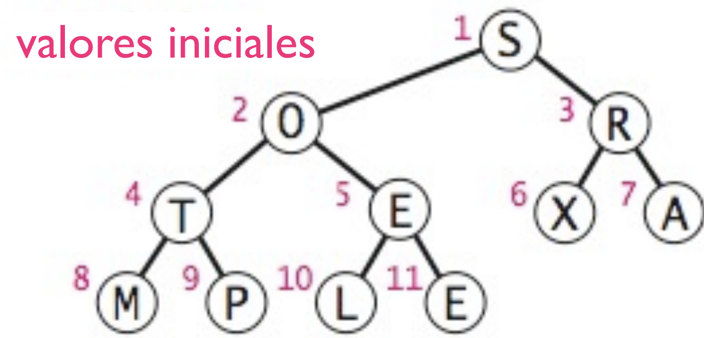
Heapsort - Creación de montículo de abajo hacia arriba

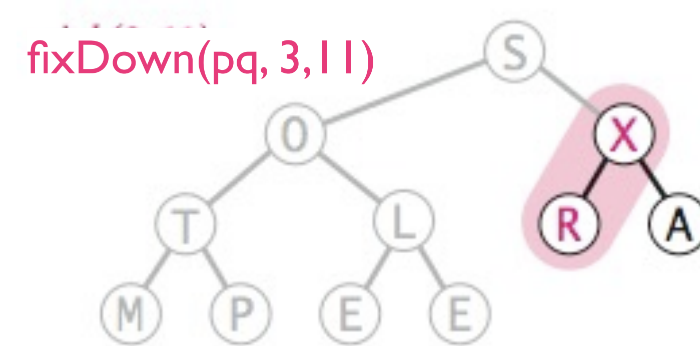


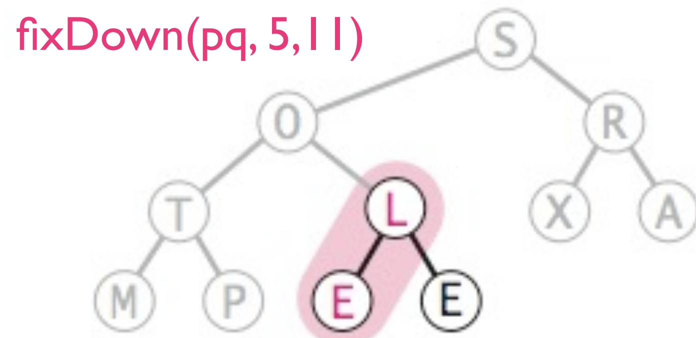


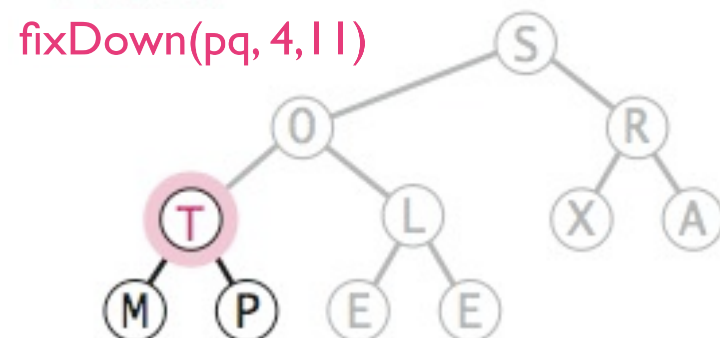


Heapsort - Creación de montículo de abajo hacia arriba

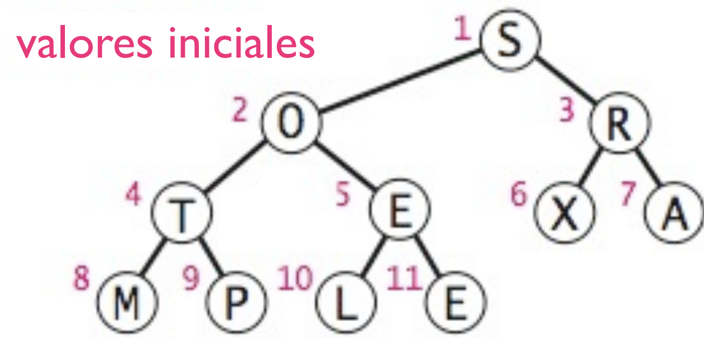


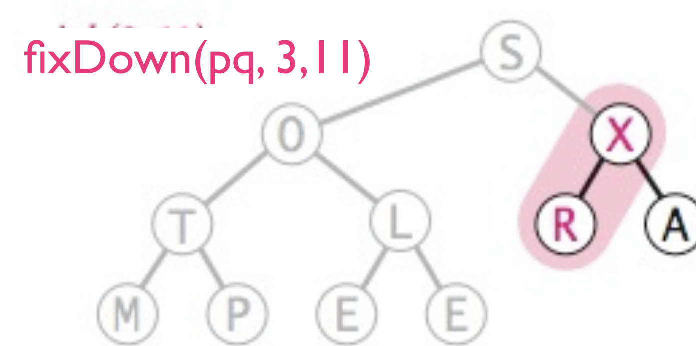


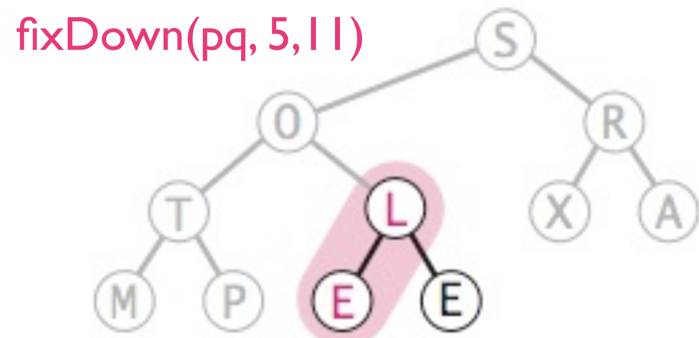


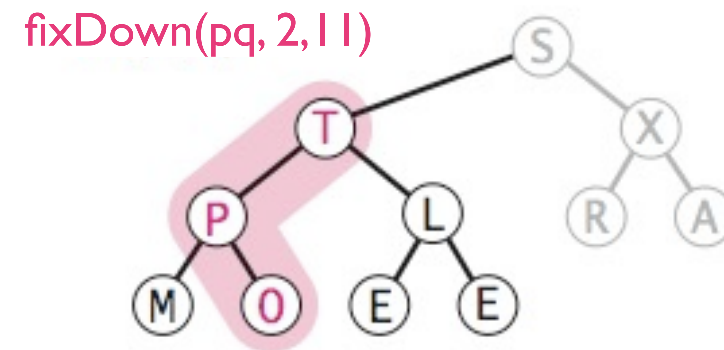


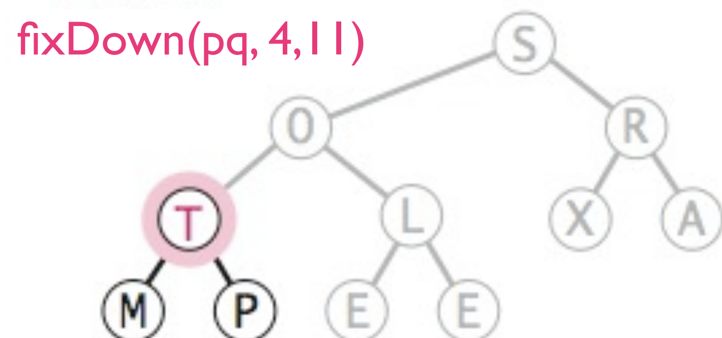
Heapsort - Creación de montículo de abajo hacia arriba



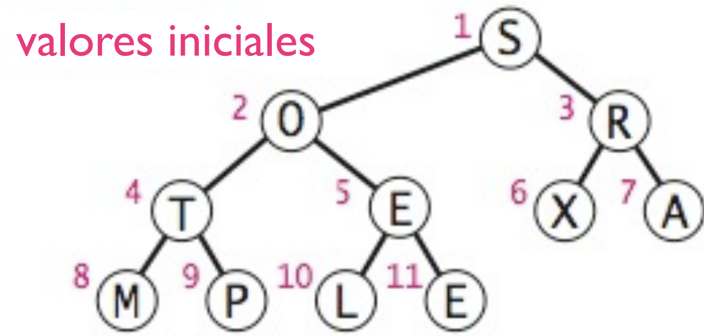


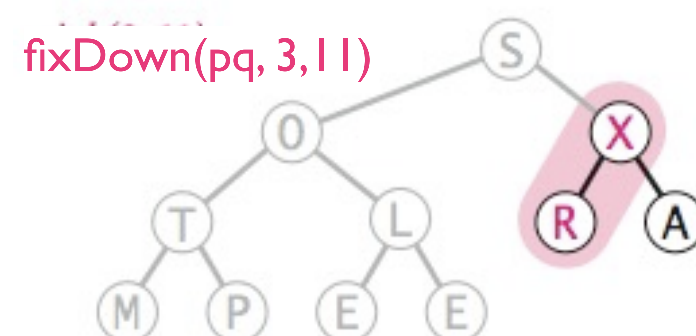


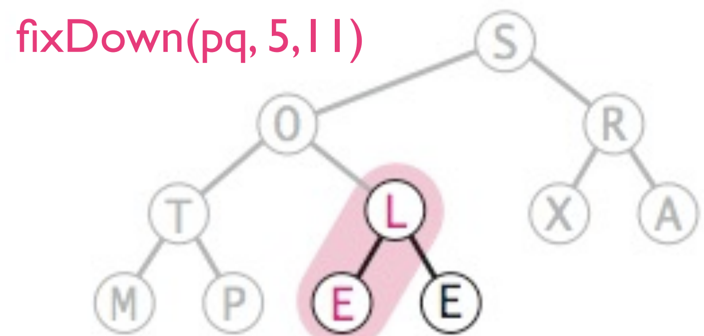


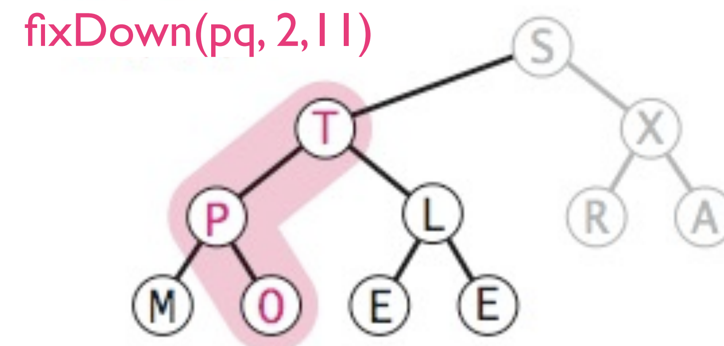


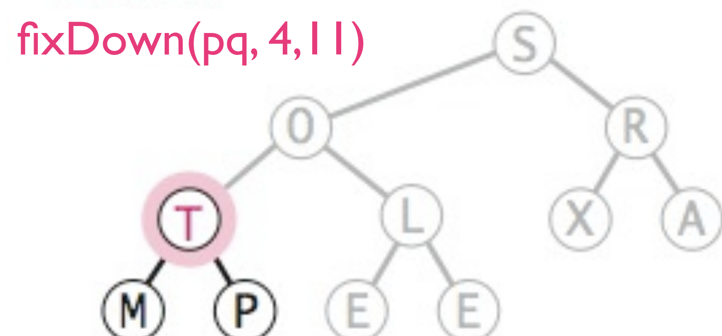
Heapsort - Creación de montículo de abajo hacia arriba

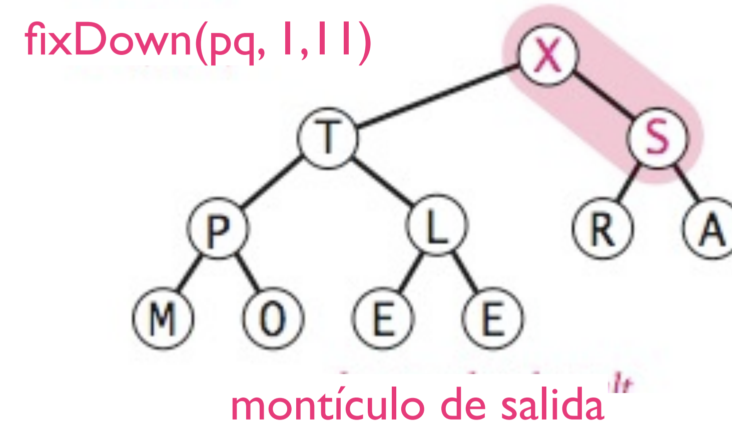












Heapsort

Heapsort

- **Segundo paso:** Ordenar

Heapsort

- **Segundo paso:** Ordenar
- **eliminar** el elemento **máximo**, uno a la vez.

Heapsort

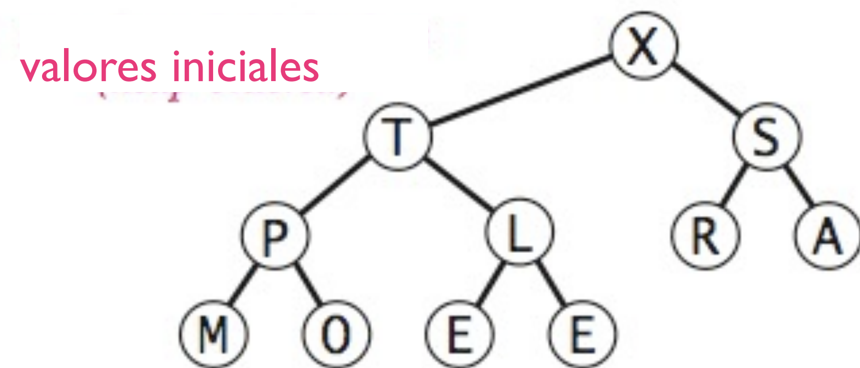
- Segundo paso: Ordenar
- eliminar el elemento máximo, uno a la vez.
- dejar el elemento en el arreglo, en lugar de "hacerlo ***null*** *" o "eliminarlo"*.

Heapsort

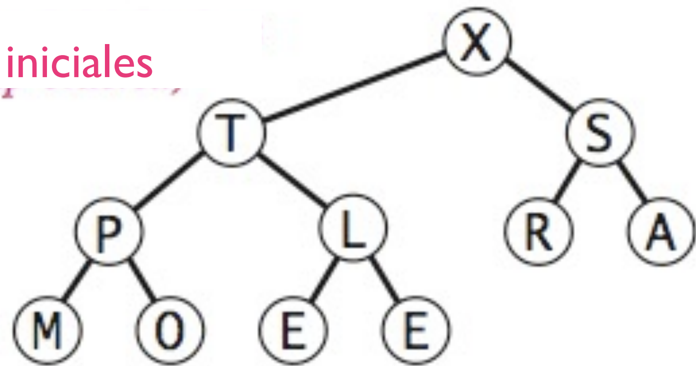
- Segundo paso: Ordenar
- eliminar el elemento máximo, uno a la vez.
- dejar el elemento en el arreglo, en lugar de "hacerlo ***null***" *o "eliminarlo"*.

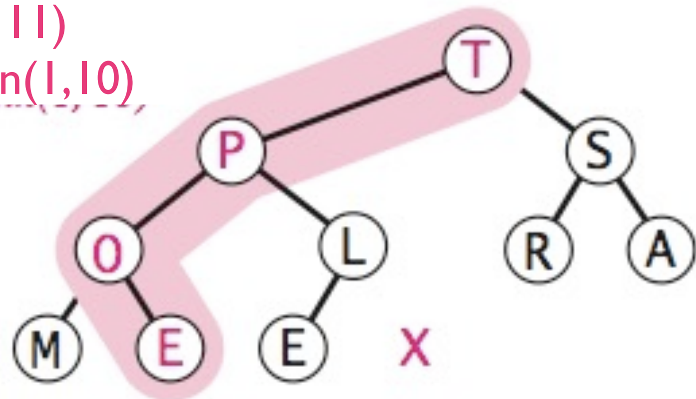
```
while ( N > 1 )
{
    exch( pq[1], pq[N] );
    fixDown( pq, 1, --N );
}
```

Heapsort - Ordenamiento

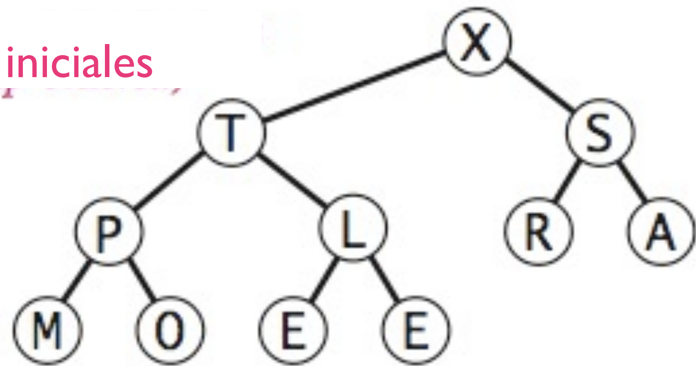


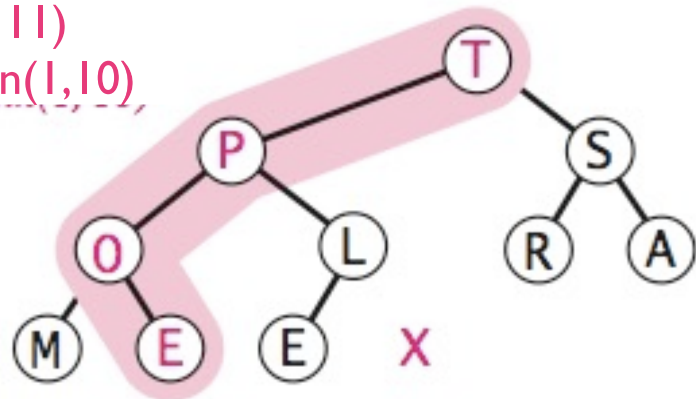
Heapsort - Ordenamiento

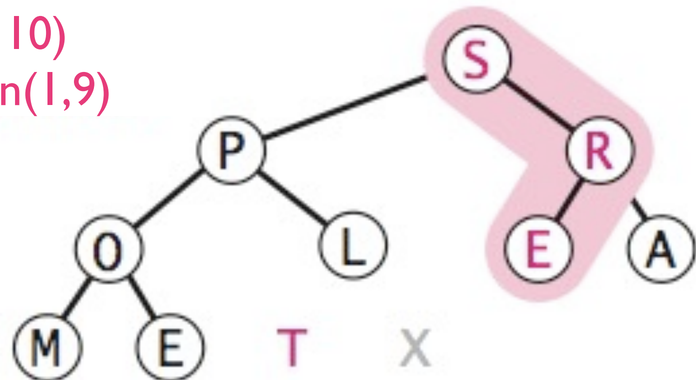




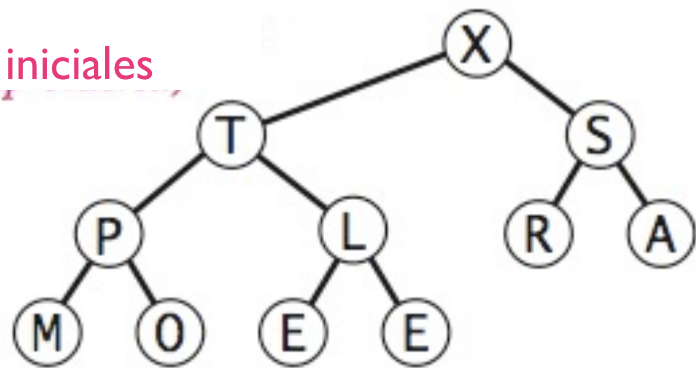
Heapsort - Ordenamiento

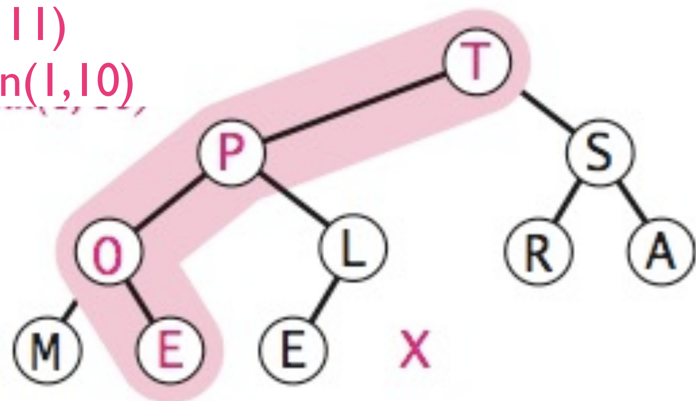


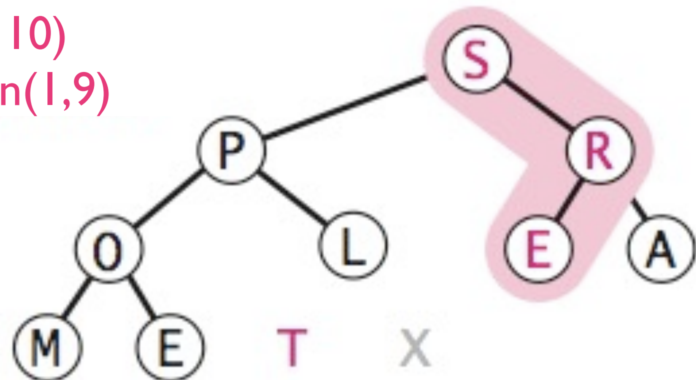


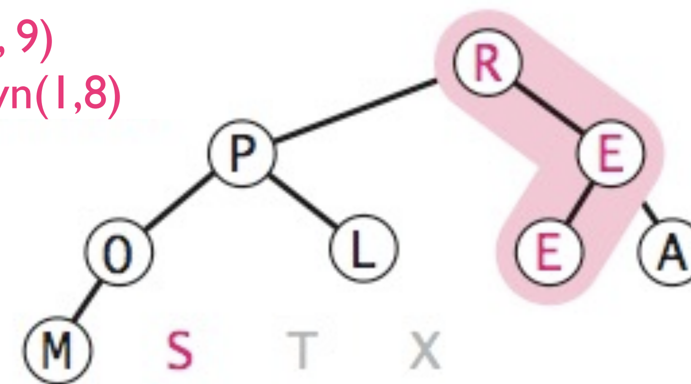


Heapsort - Ordenamiento

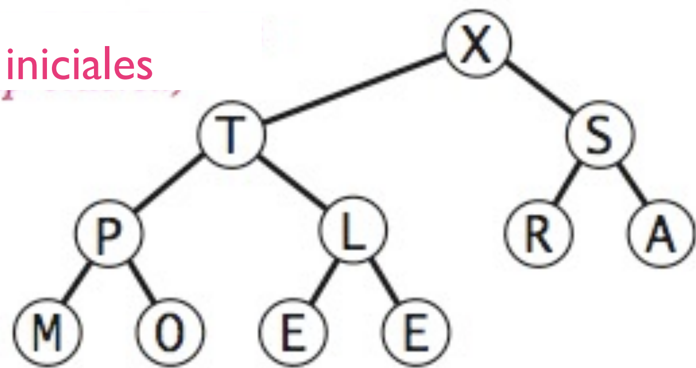


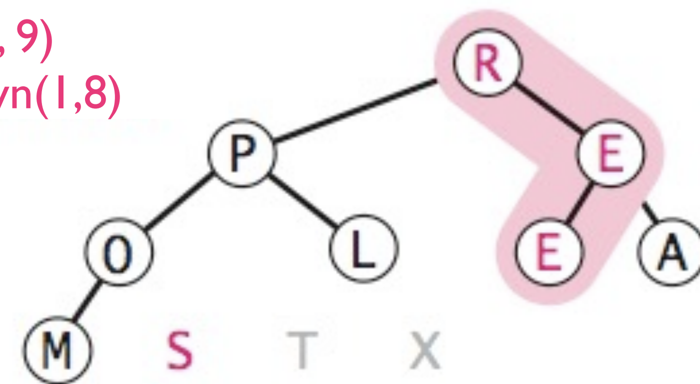


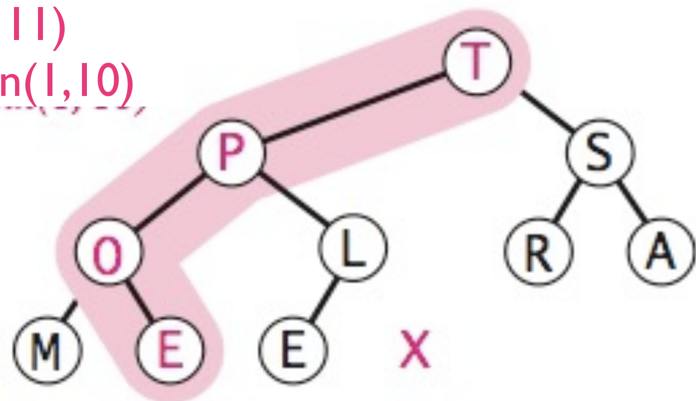


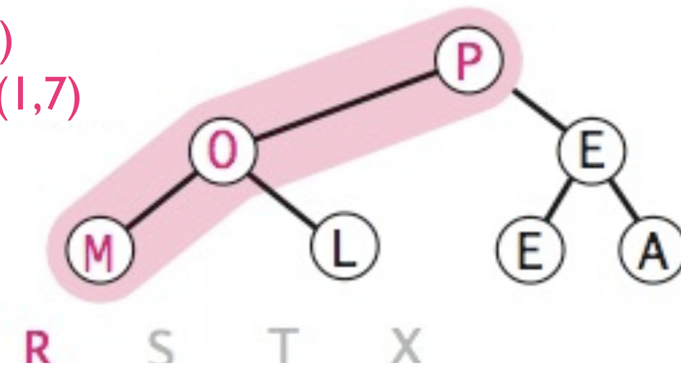


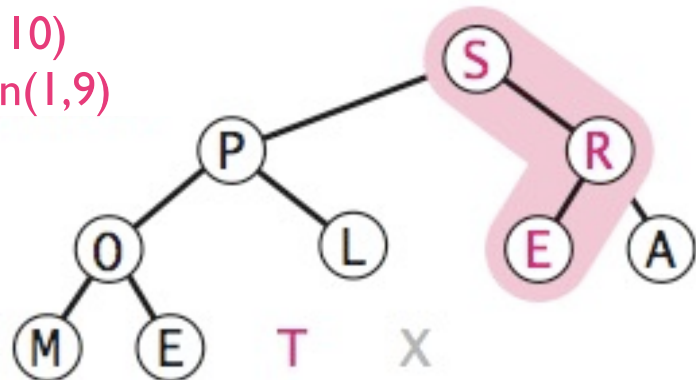
Heapsort - Ordenamiento



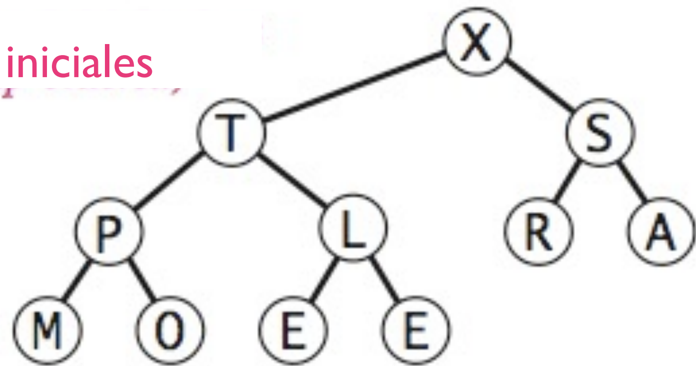


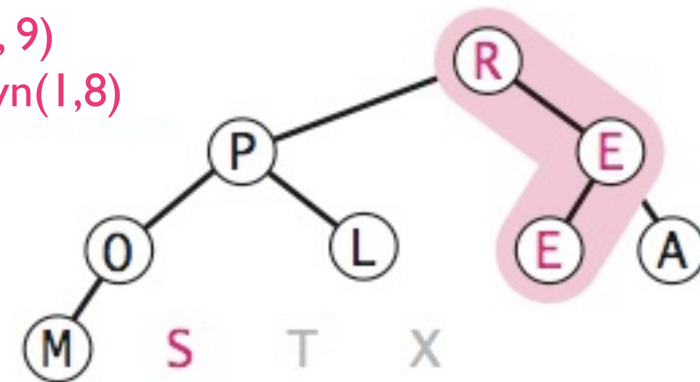


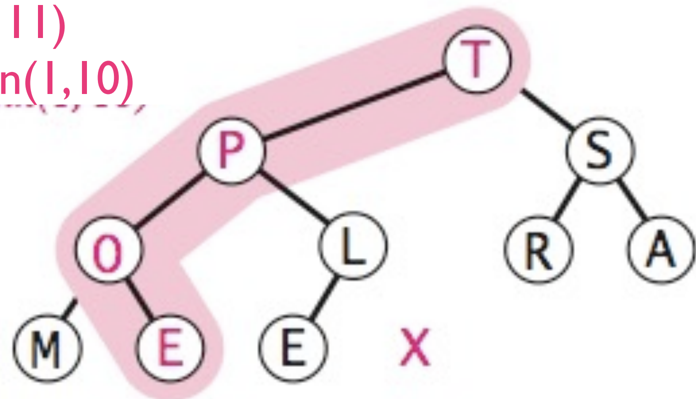


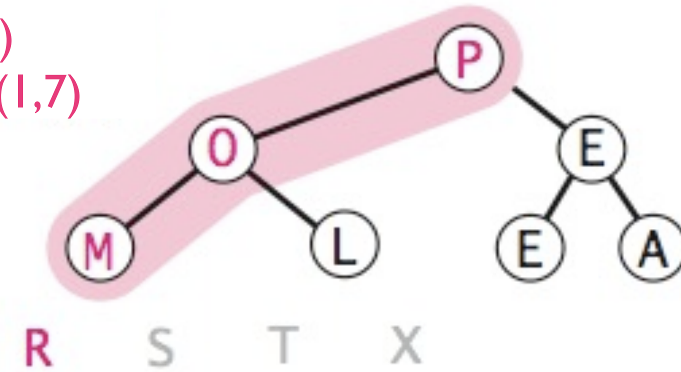


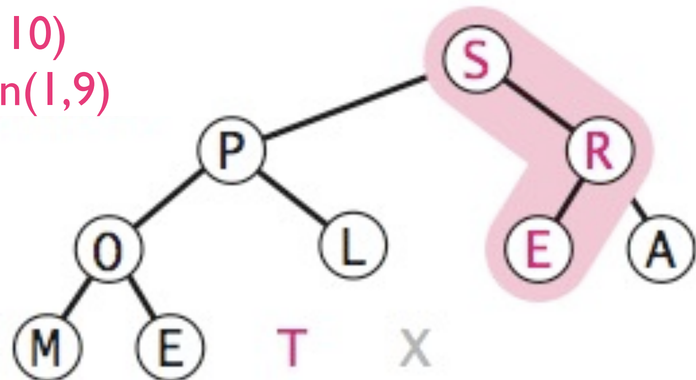
Heapsort - Ordenamiento

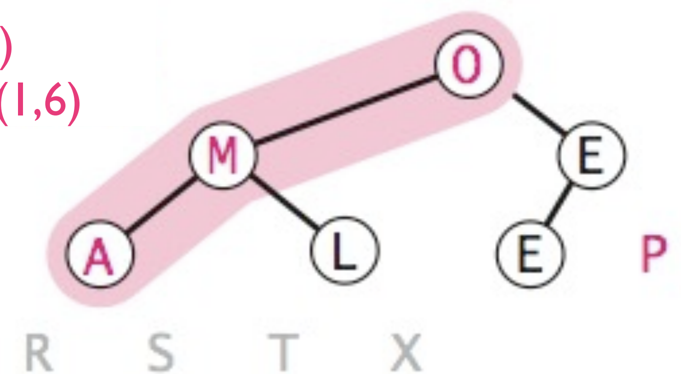




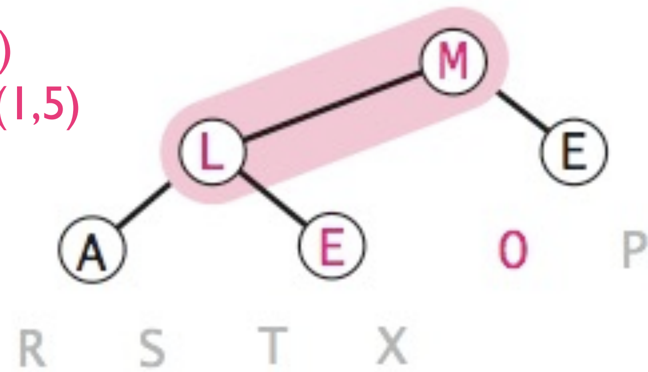






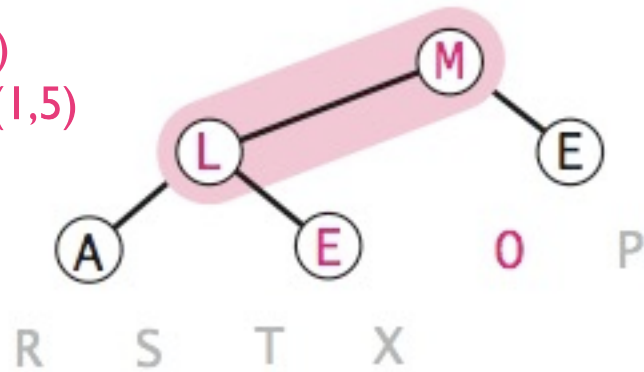


Heapsort - Ordenamiento

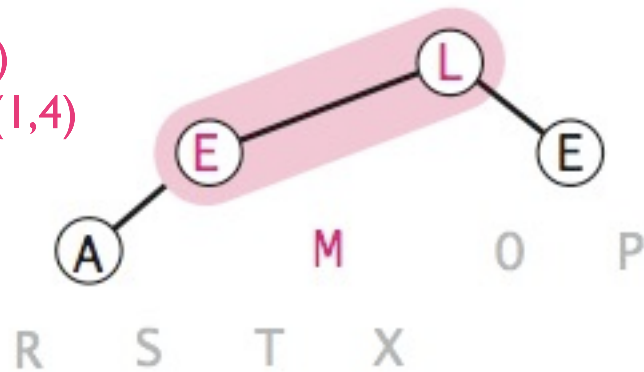


Heapsort - Ordenamiento

exch(1, 6)
fixDown(1,5)

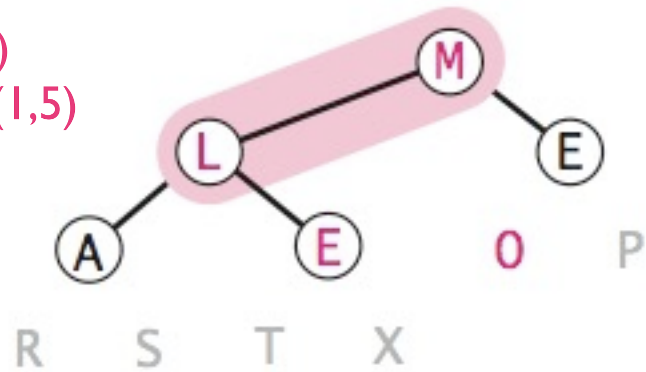


exch(1, 5)
fixDown(1,4)

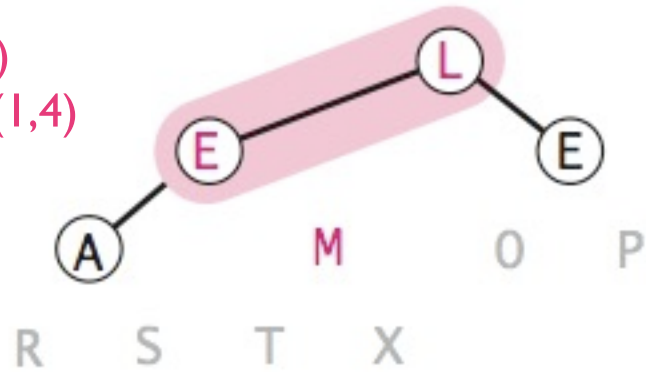


Heapsort - Ordenamiento

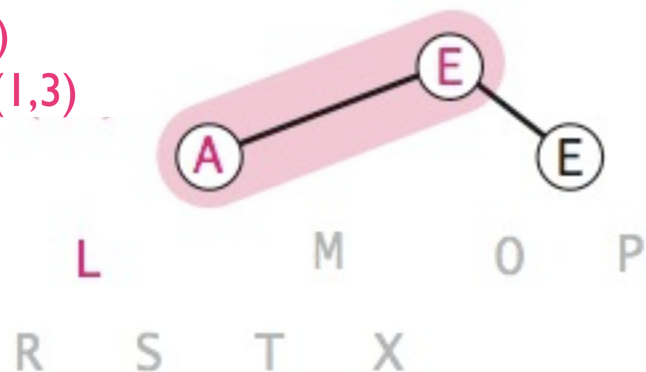
exch(1, 6)
fixDown(1,5)



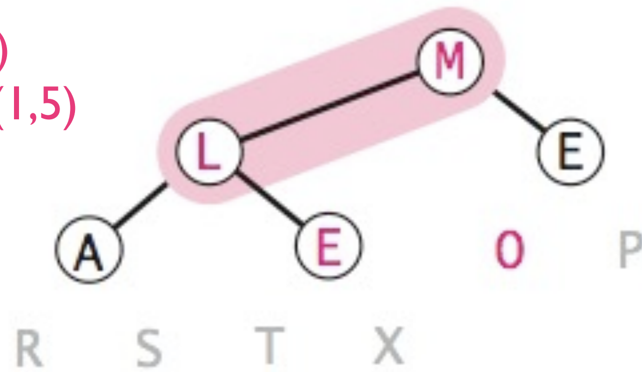
exch(1, 5)
fixDown(1,4)

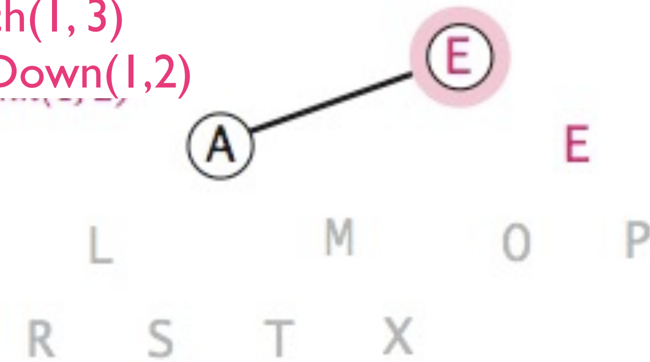


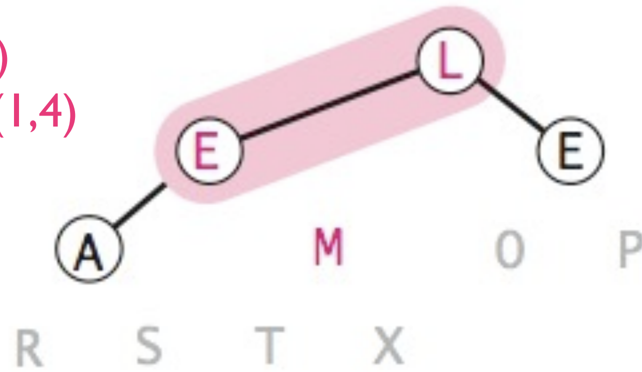
exch(1, 4)
fixDown(1,3)

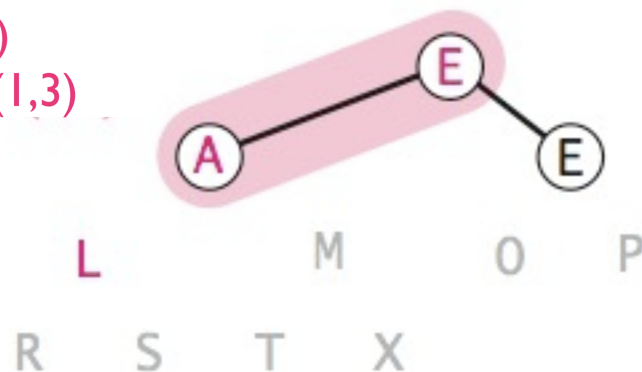


Heapsort - Ordenamiento

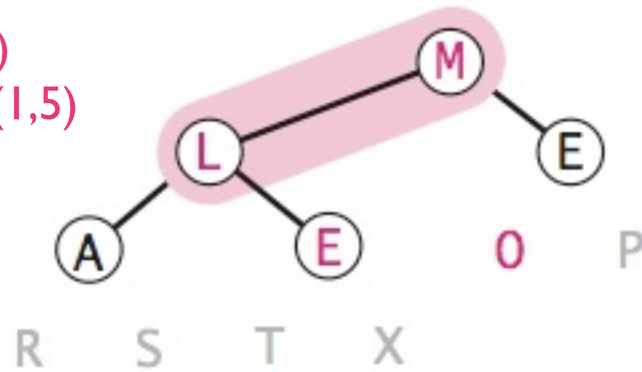


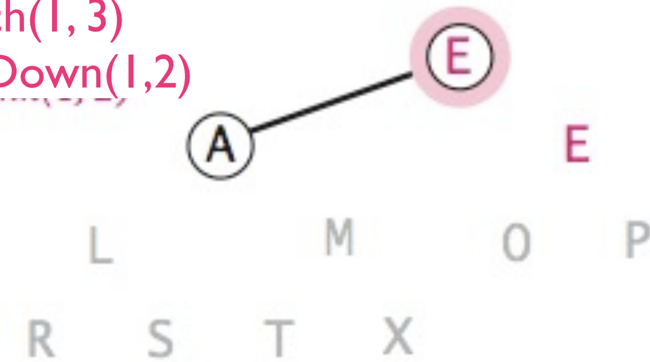


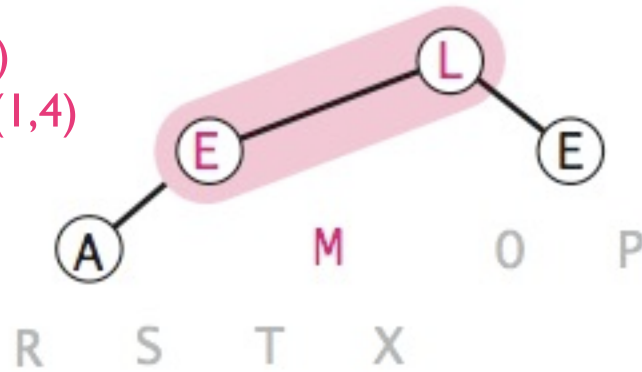


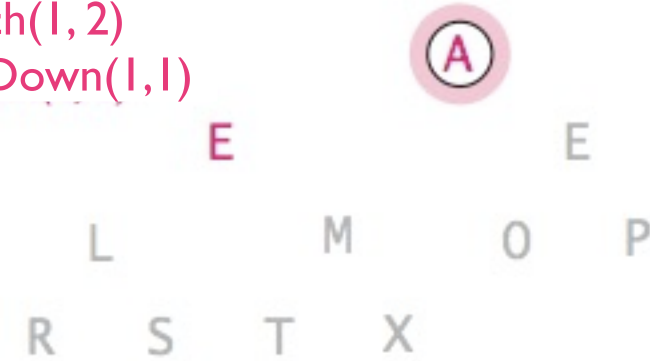


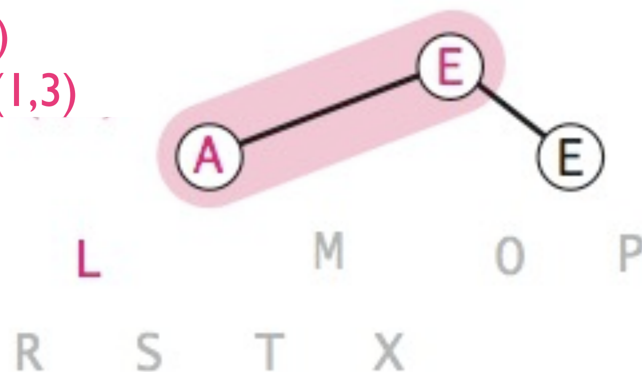
Heapsort - Ordenamiento



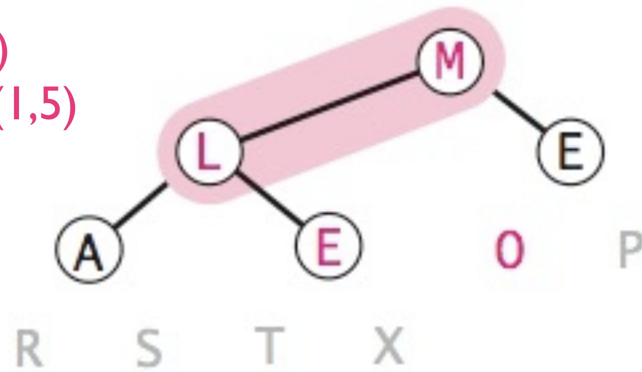


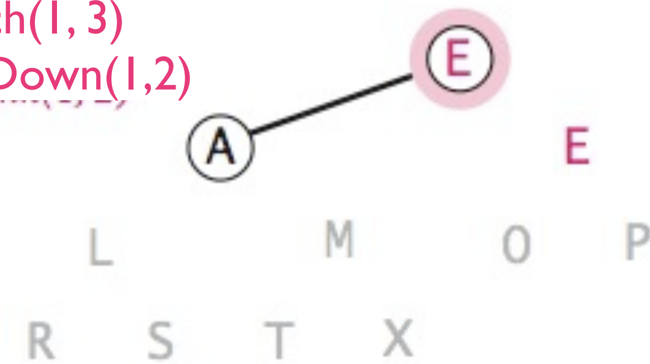


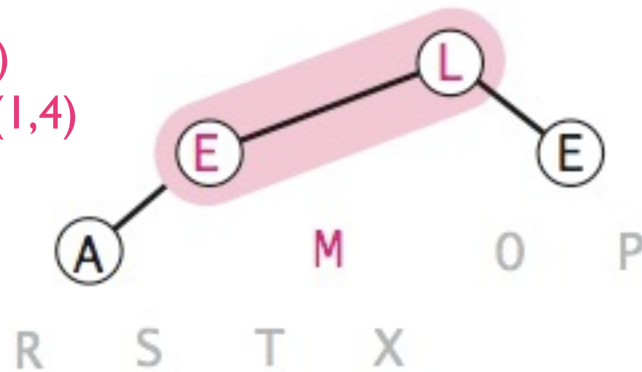


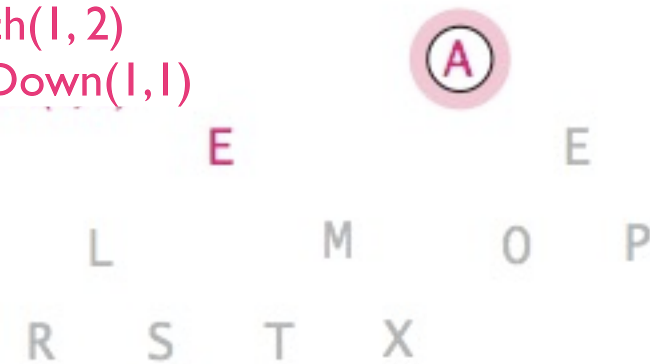


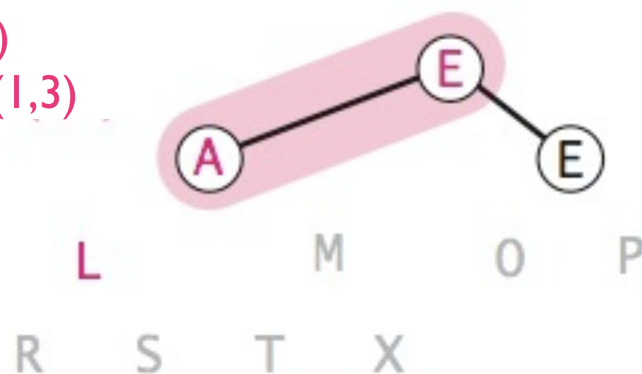
Heapsort - Ordenamiento

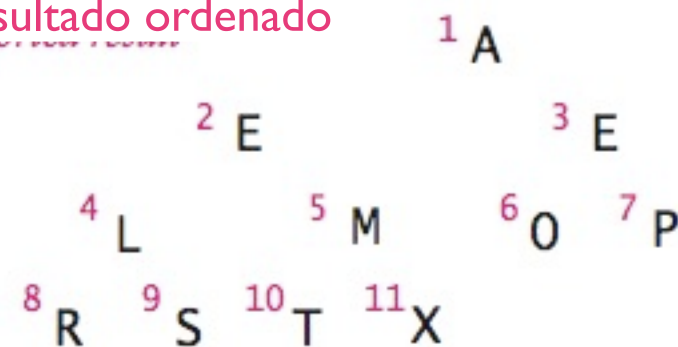






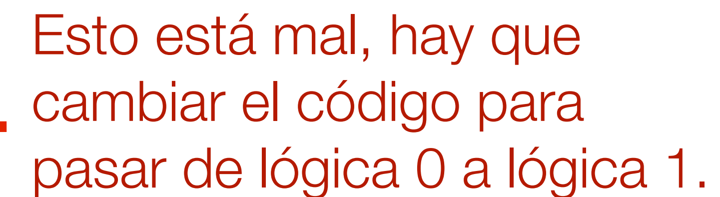






Heapsort

```
void heapSort( Item a[], int l, int r )
{
    int k, N=r-l+1;
    Item *pq = a+l ;
    // construccion del monticulo
    for( int k=N/2; k>=1; k-- )
        fixDown( pq, k, N );
    // intercambia el elemento mas grande
    // con el ultimo nodo y arregla el
    // monticulo
    while ( N > 1 )
    {
        exch( pq[1], pq[N] );
        fixDown( pq, 1, --N );
    }
}
```



Heapsort

N	k	0	1	2	3	4	5	6	7	8	9	10	11
valores iniciales			S	O	R	T	E	X	A	M	P	L	E
11	5		S	O	R	T	L	X	A	M	P	E	E
11	4		S	O	R	T	L	X	A	M	P	E	E
11	3		S	O	X	T	L	R	A	M	P	E	E
11	2		S	T	X	P	L	R	A	M	O	E	E
11	1		X	T	S	P	L	R	A	M	O	E	E
ordenados en montículo			X	T	S	P	L	R	A	M	O	E	E
10	1		T	P	S	O	L	R	A	M	E	E	X
9	1		S	P	R	O	L	E	A	M	E	T	X
8	1		R	P	E	O	L	E	A	M	S	T	X
7	1		P	O	E	M	L	E	A	R	S	T	X
6	1		O	M	E	A	L	E	P	R	S	T	X
5	1		M	L	E	A	E	O	P	R	S	T	X
4	1		L	E	E	A	M	O	P	R	S	T	X
3	1		E	A	E	L	M	O	P	R	S	T	X
2	1		E	A	E	L	M	O	P	R	S	T	X
1	1		A	E	E	L	M	O	P	R	S	T	X
resultado ordenado			A	E	E	L	M	O	P	R	S	T	X

los elementos en negro son comparados

los elementos en rojo son intercambiados

los elementos en gris no se modifican

Salida del algoritmo heapsort (contenido del arreglo después de cada fixDown)

Heapsort - Desempeño

Heapsort - Desempeño

- Propiedad 1.

Heapsort - Desempeño

- Propiedad 1.
    - La construcción de montículos de abajo hacia arriba toma tiempo lineal.

Heapsort - Desempeño

- Propiedad 1.
  - La construcción de montículos de abajo hacia arriba toma tiempo lineal.

Heapsort - Desempeño

- Propiedad 1.
  - La construcción de montículos de abajo hacia arriba toma tiempo lineal.

Heapsort - Desempeño

- Propiedad 1.
    - La construcción de montículos de abajo hacia arriba toma tiempo lineal.

- Por ejemplo, construir un montículo de 127 elementos, procesamos

Heapsort - Desempeño

- Propiedad 1.
    - La construcción de montículos de abajo hacia arriba toma tiempo lineal.

- Por ejemplo, construir un montículo de 127 elementos, procesamos
- 32 montículos de tamaño 3, 16 montículos de tamaño 7, 8 montículos de tamaño 15, 4 montículos de tamaño 31, 2 montículos de tamaño 63 y 1 montículo de tamaño 127:

Heapsort - Desempeño

- Propiedad 1.
  - La construcción de montículos de abajo hacia arriba toma tiempo lineal.

- Por ejemplo, construir un montículo de 127 elementos, procesamos
  - 32 montículos de tamaño 3, 16 montículos de tamaño 7, 8 montículos de tamaño 15, 4 montículos de tamaño 31, 2 montículos de tamaño 63 y 1 montículo de tamaño 127:
  - (32) + (16)(2) + (8)(3) + (4)(4) + (2)(5) + (1)(6) = 120 degradaciones

Heapsort - Desempeño

- Propiedad 1.
    - La **construcción** de montículos **de abajo hacia arriba** toma tiempo **lineal**.

- Por ejemplo, construir un montículo de 127 elementos, procesamos
  - 32 montículos de tamaño 3, 16 montículos de tamaño 7, 8 montículos de tamaño 15, 4 montículos de tamaño 31, 2 montículos de tamaño 63 y 1 montículo de tamaño 127:
  - (32) + (16)(2) + (8)(3) + (4)(4) + (2)(5) + (1)(6) = 120 degradaciones
  - el doble de comparaciones para el peor caso.

Heapsort - Desempeño

Heapsort - Desempeño

- El **tiempo de cálculo** para un algoritmo Heapsort está **dominado** por el **ordenamiento hacia abajo** O(n log n) y no por la construcción O(n).

Heapsort - Desempeño

- El tiempo de cálculo para un algoritmo Heapsort está dominado por el ordenamiento hacia abajo O(n log n) y no por la construcción O(n).

Heapsort - Desempeño

- El tiempo de cálculo para un algoritmo Heapsort está dominado por el ordenamiento hacia abajo O(n log n) y no por la construcción O(n).

- Propiedad 2.

Heapsort - Desempeño

- El tiempo de cálculo para un algoritmo Heapsort está dominado por el ordenamiento hacia abajo O(n log n) y no por la construcción O(n).

- Propiedad 2.

  - El ordenamiento con montículos usa menos de ***2 N lg N*** comparaciones para ordenar ***N*** elementos.

Heapsort - Comparación

Heapsort - Comparación

- Heapsort garantiza ordenar **N** elementos en su lugar en un tiempo proporcional a **N log N** sin importar la entrada.

Heapsort - Comparación

- Heapsort garantiza ordenar **N** elementos en su lugar en un tiempo proporcional a **N log N** sin importar la entrada.

Heapsort - Comparación

- Heapsort garantiza ordenar **N** elementos en su lugar en un tiempo proporcional a **N log N** sin importar la entrada.

- No hay entrada de peor caso que haga el algoritmos Heapsort significativamente más lento (como es el caso de Quicksort).

Heapsort - Comparación

- Heapsort garantiza ordenar **N** elementos en su lugar en un tiempo proporcional a **N log N** sin importar la entrada.

- No hay entrada de peor caso que haga el algoritmos Heapsort significativamente más lento (como es el caso de Quicksort).

- No utiliza espacio adicional (como es el caso de Mergesort).

Heapsort - Comparación

- Heapsort garantiza ordenar **N** elementos en su lugar en un tiempo proporcional a **N log N** sin importar la entrada.

- No hay entrada de peor caso que haga el algoritmos Heapsort significativamente más lento (como es el caso de Quicksort).
- No utiliza espacio adicional (como es el caso de Mergesort).
- El ciclo interno (costo por comparación) tiene más operaciones básicas que Quicksort y utiliza más comparaciones que Quicksort para entradas aleatorias.

Heapsort - Comparación

- Heapsort garantiza ordenar **N** elementos en su lugar en un tiempo proporcional a **N log N** sin importar la entrada.

- No hay entrada de peor caso que haga el algoritmos Heapsort significativamente más lento (como es el caso de Quicksort).
- No utiliza espacio adicional (como es el caso de Mergesort).
- El ciclo interno (costo por comparación) tiene más operaciones básicas que Quicksort y utiliza más comparaciones que Quicksort para entradas aleatorias.
- Heapsort también es útil para problemas de selección, como encontrar el ***k*** elemento mayor entre ***N*** elementos (deteniendo el algoritmo después de ***k*** extracciones).

Heapsort - Comparación

- Ordenar en ***N lg N*** en el peor caso sin necesidad de memoria extra.
  - Mergesort: no, espacio extra lineal.
  - Quicksort: no, peor caso de tiempo cuadrático.
  - Heapsort: si.
- Heapsort es óptimo en tiempo y espacio pero:
  - el ciclo interno es más largo que en quicksort
  - uso extensivo de la memoria cache.

Heapsort - Comparación

	inplace?	stable?	worst	average	best	remarks
selection	×		$N^2/2$	$N^2/2$	$N^2/2$	N exchanges
insertion	×	×	$N^2/2$	$N^2/4$	N	use for small N or partially ordered
shell	×		?	?	N	tight code, subquadratic
quick	×		$N^2/2$	$2N\ln N$	$N\lg N$	$N\lg N$ probabilistic guarantee fastest in practice
3-way quick	×		$N^2/2$	$2N\ln N$	$N\lg N$	improves quicksort in presence of duplicate keys
merge		×	$N\lg N$	$N\lg N$	$N\lg N$	$N\lg N$ guarantee, stable
heap	×		$2N\lg N$	$2N\lg N$	$N\lg N$	$N\log N$ guarantee, in-place
???	×	×	$N\lg N$	$N\lg N$	$N\lg N$	holy sorting grail

Heapsort

Heapsort